Francesco Balena

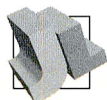
I SEGRETI DI MICROSOFT BASIC

Per Microsoft QuickBASIC 4.x, BASIC PDS 7.x e Visual Basic per DOS

Tecniche di ottimizzazione per programmi più veloci e compatti

Tutte le caratteristiche non documentate

Collegamento con il linguaggio Assembly

include disco

JACKSON LIBRI

I SEGRETI DI MICROSOFT BASIC

Francesco Balena

Via Rosellini, 12 - 20124 Milano

INDICE GENERALE

Ad Arianna

RINGRAZIAMENTI

Questo libro non sarebbe mai stato pubblicato - non in questa forma almeno - senza la collaborazione di Alessandro Pedone, direttore della rivista *Computer Programming* della Infomedia, che ha gentilmente acconsentito alla pubblicazione di alcuni articoli che avevo preparato a suo tempo per la rivista.

Questa è anche l'occasione per ringraziare Mario Lombardini della Softmedia-Dr.Awing, infaticabile mago della computer graphic, per la creatività, l'abilità e la pazienza necessarie per concretizzare sullo schermo e sulla carta le mie nebulose intuizioni cetaceo-informatiche; il logo della SoftWhale che potete ammirare in quarta di copertina è frutto della sua matita, anzi del suo mouse.

Continuando nel filone "questo libro non sarebbe mai stato pubblicato senza l'aiuto di...", non posso non ricordare Giuseppe Ippolito e gli altri amici della Software Design S.r.l.: ad una settimana dalla improrogabile consegna delle bozze il mio computer è andato in malora portando con sé word processor e tutti i file *.DOC, e senza il "pronto intervento" del suddetto starei mettendo ancora a soqquadro il mio studio alla ricerca di qualche improbabile dischetto di backup.

INTRODUZIONE

Ogni libro ha la sua storia, e quello che state leggendo non fa eccezione. Già da alcuni anni pensavo che avrei voluto scrivere un testo che facesse piazza pulita dei soliti luoghi comuni e fornisse la possibilità di approfondire alcuni aspetti della programmazione BASIC anche a coloro che già usano più o meno professionalmente questo linguaggio. Dopo tanti impegni di lavoro, finalmente sono riuscito a ritagliare qualche mese per raccogliere tutte (o quasi) le mie conoscenze sul Microsoft BASIC - accumulate in oltre sei anni di lavoro quotidiano - e metterle in una forma intelleggibile anche ad altri.

A differenza della quasi totalità dei libri pubblicati sull'argomento, questo testo non è un "tutorial" sul linguaggio, poiché darò per scontato che conosciate già i rudimenti del linguaggio, e non è neanche un "reference" delle istruzioni, perché non mi interessa riscrivere i manuali in dotazione al linguaggio, che peraltro sono scritti molto bene nella migliore tradizione Microsoft (a parte qualche omissione e imprecisione che avrò modo di illustrare nei vari capitoli). Volendo a tutti i costi definire questo lavoro, mi piacerebbe considerarlo un insieme di "appunti di lavoro", in cui ho annotato tante piccole scoperte su come risolvere i problemi più comuni o migliorare le prestazioni del linguaggio nei campi in cui è più carente. Piuttosto che ripetere cose già scritte altrove, ho preferito evitare di trattare argomenti che non ho avuto modo o occasione di approfondire; per questo motivo non troverete un capitolo dedicato alla grafica o alla gestione di basi di dati su disco. Per non penalizzare gli utenti delle precedenti versioni del BASIC - che

a tutt'oggi formano la maggioranza dei programmatori BASIC - ho evitato di soffermarmi sulle caratteristiche peculiari del nuovo Visual BASIC per Dos e in generale della programmazione orientata agli eventi: con pochissime eccezioni tutte le tecniche descritte possono essere applicate a qualunque versione di BASIC sotto Dos a partire dal QuickBASIC 4.0, e molte considerazioni possono essere considerate valide persino per chi sviluppa in Visual BASIC per Windows.

Un'ultima considerazione: in generale evito di perdere tempo in interminabili discussioni su quale sia "il miglior linguaggio" oppure quale compilatore produca il codice più ottimizzato, e così via. Personalmente trovo che il BASIC sia un linguaggio estremamente produttivo, grazie alla combinazione interprete+compilatore, e che produca programmi discretamente veloci e compatti; forse con un altro linguaggio potrei ottenere programmi un po' più veloci, ma molto probabilmente la loro stesura richiederebbe più tempo; dovendo scegliere tra sprecare il mio tempo o quello di un computer preferisco sempre la seconda soluzione.

BREVE STORIA DEL MICROSOFT BASIC

Per quel che riguarda i personal computer Ms-Dos, già sul primo modello di PC IBM era disponibile un interprete denominato BASICA; la sua caratteristica principale era di risiedere in parte su disco e in parte su ROM, e di essere attivato automaticamente al bootstrap, nel caso in cui l'operatore non faceva trovare un dischetto contenente il sistema operativo nel drive A: (i dischi rigidi erano rari, a questi tempi). La scelta di memorizzare il BASIC su ROM fu in parte un residuo dell'era degli home computer, e in parte una necessità per facilitare il lavoro dei programmatori, tenuto conto che all'epoca la quasi totalità dei programmi in circolazione erano scritti in Assembly e - appunto - in BASIC.

Con l'apparizione dei primi cloni compatibili Ms-Dos, i costruttori di computer si trovarono nell'impossibilità di copiare il BASICA, che era sotto il copyright di IBM, e dovettero ricorrere ad un interprete caricato da disco, il GW-BASIC. A parte il nome e qualche istruzione, le due versioni di BASIC erano praticamente identiche, al punto che la maggior parte dei programmi potevano essere fatti girare indifferentemente sotto l'uno o l'altro interprete; ovviamente non si trattava di un caso fortuito, dato gli autori dei due linguaggi erano gli stessi, ovvero Bill Gates, Greg Whitten e qualcun'altro. Queste prime versioni del BASIC erano a davvero misere: i numeri di linea erano obbligatori, non esistavano procedure e funzioni né variabili locali; le stringhe erano limitate a 255 caratteri ed esistevano notevoli limiti alle dimensioni dei programmi e dei dati. A conti fatti, queste prime versioni hanno fatto tantissimo per la

diffusione del linguaggio, ma hanno anche contribuito a una serie di equivoci sulla natura del linguaggio: provate a chiedere - oggi - ad un programmatore C, FORTRAN o COBOL cosa ne pensa del BASIC e quattro volte su cinque vi sentirete rispondere che "..non è possibile lavorare con un linguaggio così primitivo, con i numeri di riga invece delle etichette e con i suffissi $,! e % per indicare i differenti tipi di variabili...": è ovvio che si riferiscono a queste prime versioni del linguaggio e che non hanno mai visto un listato BASIC prodotto negli ultimi cinque anni, o se lo hanno visto lo hanno scambiato per Pascal !

Ad ogni modo, è stato necessario attendere dieci anni per vedere qualcosa di nuovo sul fronte degli interpreti "puri" del BASIC; solo con l'introduzione della versione 5 di Ms-Dos un nuovo e potente interprete fu incluso nel sistema operativo: il QBasic, che non è altro che la versione interpretata del QuickBASIC 4.5; ovviamente la Microsoft aveva fatto bene i suoi conti, prevedendo giustamente che molti di coloro che avessero trovato interessante il QBasic avrebbero poi acquistato il prodotto completo.

Sul fronte dei compilatori, per fortuna, l'evoluzione è stata continua e anche ragionevolmente veloce. Il primo compilatore per BASIC fu rilasciato da IBM all'inizio del 1982, e fu chiamato BASCOM 1.0 (da BAS*ic* COM*piler*, ovviamente), anche se, come per il BASICA, si trattava di un prodotto sviluppato in realtà da Microsoft. Rispetto agli interpreti BASICA e GW-BASIC, il compilatore includeva numerose novità, prima tra tutte la facoltà di evitare i numeri di riga (ad eccezione delle righe da etichettare come destinazioni di GOTO e GOSUB) e la possibilità di assegnare fino a 32K caratteri ad una stringa. La versione successiva fu rilasciata da IBM nel 1985: il BASCOM 2.0 prevedeva le etichette alfanumeriche, gli array dinamici, le procedure richiamabili per nome e i moduli, che permettevano finalmente di superare il limite dei 64K per il codice.

Poco dopo l'introduzione del BASCOM 2, la Microsoft entrò direttamente sul mercato proponendo un prodotto simile, denominato per l'occasione QuickBASIC 1.0; anche se le caratteristiche erano in un certo senso inferiori, il prezzo allettante ne decretò immediatamente il successo, tanto che l'anno dopo fu rilasciato il QuickBASIC 2, che finalmente presentava il primo ambiente integrato e la possibilità di utilizzare le Quick Libraries. Nel 1987 arrivò il QuickBASIC 3.0, che includeva il primo debugger integrato, e alcune migliorie al linguaggio, tra cui i cicli DO e le istruzioni SELECT CASE, e il supporto per il coprocessore matematico; questa è anche la prima versione del compilatore col doppio supporto per le routine IEEE per i calcoli floating point e per le routine proprietarie di Microsoft, più veloci ma meno precise: questo doppia possibilità veniva ottenuta usando due differenti compilatori.

Il grande balzo in avanti si ebbe con il QuickBASIC 4.0 (fine 1987), il primo a supportare l'ambiente di sviluppo con tecnologia *p-code* e un debugger degno di questo nome (con breakpoint, stepping, espressioni di watch, ecc.); la lista

di miglioramenti al linguaggio è lunghissima, ma vale la pena di ricordare le variabili strutturate (TYPE), gli array huge, le stringhe di lunghezza fissa, le FUNCTION (che si affiancavano alle pre-esistenti SUB), il passaggio di argomenti per indirizzo oltre che per valore, il supporto per CodeView ed altri ancora. Subito dopo la Microsoft rilasciò il BASCOM 6.0, in tutto e per tutto simile al QuickBASIC 4.0, ma che supportava entrambe le librerie matematiche (il QB si limitava alle sole IEEE, più lente delle altre), includeva un maggior numero di *stub files* e permetteva di creare eseguibili per OS/2 (solo per il modo testo). Il QuickBASIC 4.5 arrivò nel 1988 e probabilmente è la versione di BASIC più diffusa a tutt'oggi; le novità rispetto alla versione precedente riguardavano prevalentemente l'ambiente di sviluppo (ad es. un sistema di help più completo e un tutorial), ma esso fornì l'occasione per ripulire il linguaggio di alcuni bug. Il QuickBASIC 4.5 è disponibile sia in italiano (programma e manuali) che in inglese.

L'anno seguente fu presentato il BASIC PDS (Professional Development System) versione 7.0; già dalla numerazione era evidente che con questo prodotto la Microsoft intendeva soppiantare il BASCOM e creare un unico ambiente per lo sviluppo "veloce" (grazie alla presenza dell'ambiente di sviluppo QBX) ma anche per le applicazioni professionali; il PDS infatti permette di usare segmenti multipli per le stringhe (stringhe far), il trattamento locale degli errori (ON LOCAL ERROR) e soprattutto un gestore ISAM e alcune librerie per la costruzione di interfacce utente evolute, per creare grafici, per la gestione del mouse. L'ambiente di sviluppo era in grado di sfruttare sia la *High Memory Area* (i primi 64K oltre il primo megabyte) che la memoria espansa, e di lasciare più memoria libera al programma sotto sviluppo. Pochi mesi dopo fu rilasciato il BASIC PDS 7.1, che introduceva l'istruzione REDIM PRESERVE, la possibilità di dichiarare degli parametri BYVAL e di passare delle stringhe fissa come argomenti a procedure; per l'occasione il gestore ISAM fu riscritto e ottimizzato, ed altri bug furono rimossi. Il BASIC PDS esiste nella sola versione inglese.

Il BASIC PDS 7.1 non è stato mai aggiornato ulteriormente per più di tre anni, una eternità in confronto alla rapidità con cui le varie versioni si erano succedute, e sembrava che la Microsoft avesse deciso di abbandonare il prodotto che l'aveva resa celebre e che avesse concentrato l'attenzione sulla nuova gallina dalle uova d'oro, il *Visual* BASIC *per Windows* che in America stava battendo tutti i record di incassi nella categoria "programming tools". Per questo motivo l'annuncio all'inizio 1993 del Visual BASIC per Dos ha colto un po' tutti di sorpresa. Il realtà, come si visto subito, questa nuova versione vorrebbe essere un ponte per i programmatori BASIC che prevedono di passare prima o poi alla versione per Windows, ed è tutto sommato la più profonda riscrittura del linguaggio dai tempi del GW-BASIC. Il VBDOS è disponibile nella versione Standard (italiana e inglese) e Professional (solo

inglese); la prima ha in un certo senso sostituito il QuickBASIC, mentre la seconda - che dispone di un buon gestore ISAM e di alcuni toolkit assenti nella Standard - sostituisce il Professional Development System.

Il nuovo BASIC è diventato un linguaggio orientato agli eventi, che permette lo sviluppo di interfacce moderne con poco sforzo e che dispone di un *form designer* per il disegno di maschere per l'inserimento dei dati che facilita non poco il lavoro dei programmatori alle prime armi, ma anche dei professionisti. Il rovescio della medaglia è che il nuovo BASIC produce eseguibili di grosse dimensioni, non dispone delle near string e crea di conseguenza programmi leggermente più lenti; in realtà, a parte l'approccio *event-driven*, il nucleo del linguaggio non si discosta di molto da quello del PDS. Per questo motivo, nonché per la necessità di manutenere il codice sviluppato per le versioni precedenti e per il generale "slittamento" dell'interesse di molti programmatori verso Windows, al momento in cui scrivo queste note il VBDOS non sembra essere molto diffuso e la maggioranza degli programmatori continua a lavorare con una delle versioni precedenti.

IL SET DI ISTRUZIONI DEL BASIC

Poiché non intendo riscrivere il manuale fornito a corredo del linguaggio, non troverete in questo capitolo la descrizione di tutte le istruzioni BASIC. Credo che sia molto più utile e interessante soffermarmi sulle sole istruzioni che presentano comportamenti e caratteristiche poco note o non documentate che possono creare problemi o, al contrario, essere usate a nostro vantaggio.

ASC

Come è noto, la funzione ASC() provoca un errore "Illegal Function Call" se l'argomento è una stringa nulla. Questo spesso costringe a inserire nel codice delle righe come la seguente:

```
IF LEN(a$) <> 0 THEN acode% = ASC(a$) ELSE acode% = 0
```

Lo stesso risultato si ottiene con una forma più compatta, anche se leggermente più lenta:

```
acode% = ASC(a$ + CHR$(0))
```

Quando occorre estrarre il codice ASCII di un carattere che non sia il primo di una stringa, la funzione ASC è usata insieme a MID$:

```
acode% = ASC(MID$(a$, index%, 1))
```

ma si ottiene una maggiore velocità e compattezza di codice in questo modo:

```
acode% = ASC(MID$(a$, index%))
```

ATN

La funzione ATN restituisce l'arco tangente di un numero, ossia il valore dell'angolo (espresso in radianti) la cui tangente è pari al valore passato come argomento. Al pari delle altre funzioni trigonometriche del BASIC, anche ATN è calcolata in singola o doppia precisione, a seconda che l'argomento sia in singola o doppia precisione; questo però non influisce sulla velocità di elaborazione, in quanto internamente la funzione è sempre calcolata in doppia precisione.

La funzione ATN può essere utile per ricavare il valore della costante (pi greco), sapendo che la tangente di /4 (cioè 45°) è pari ad uno:

```
pi! = ATN(1) * 4           ' risultato 3.141593
pi# = ATN(1#) * 4          ' risultato 3.14159265358979
```

BEEP

Lo stesso effetto di questa istruzione può essere ottenuto con

```
PRINT CHR$(7)
```

BLOAD, BSAVE

L'uso più comune di queste istruzioni è di scrivere e leggere delle videate in modo testo o grafico:

```
DEF SEG = &HB800              ' segmento della memoria video in modo testo
BSAVE "prova.scr", 0, 4000    ' salvataggio di una videata (4000 byte)
DEF SEG                       ' (è sempre preferibile resettare DEF SEG)
```

è altrettanto semplice, però, salvare e ricaricare i dati di un vettore o matrice, purché si calcoli con precisione il numero dei byte complessivi (usare interi LONG per evitare errori di overflow)

```
numBytes& = UBOUND(array!) * 4    ' salva un vettore di single
DEF SEG = VARSEG(array!(1))       ' (4 è la dimensione di ciascun
BSAVE "array.dat", 0, numBytes&   ' elemento)
DEF SEG
```

Nel rileggere i file salvati con BSAVE si tenga presente che, omettendo il secondo argomento (l'offset) viene usato per default il valore utilizzato in fase di salvataggio:

```
DEF SEG = VARSEG(array!(1))
BLOAD "array.dat"
DEF SEG
```

Attenzione: non è possibile salvare e ricaricare in questo modo array di lunghezza minore o pari a 16K all'interno dell'ambiente di sviluppo QBX o VBDOS, nel caso in cui si utilizzi l'opzione /Ea che forza l'allocazione di questi vettori in memoria espansa.

CALL

Se il programma sotto sviluppo contiene delle chiamate a procedure contenute in una Quick Library, è necessario che all'inizio del modulo sia presente una istruzione DECLARE SUB che informa l'interprete dell'esistenza di una routine con quel particolare nome, e di quanti e quali argomenti prevede. Queste dichiarazioni, d'altra parte, tendono ad occupare memoria e a ridurre lo spazio a disposizione per le istruzioni eseguibili. Il problema si può ridurre chiamando le procedure con il comando CALL, ad esempio

```
' se la procedura "DisegnaQuadrato" è contenuta in una
' Quick Library, le seguenti righe sono equivalenti
' ma nel secondo caso non è necessaria una DECLARE
DisegnaQuadrato riga, colonna, lato
CALL DisegnaQuadrato (riga, colonna, lato)
```

Si noti che usando CALL la lista degli argomenti deve essere racchiusa tra parentesi; se si adotta questo metodo occorre porre molta attenzione al numero e al tipo degli argomenti passati alla procedura, poiché il BASIC non sarà in grado di effettuare alcun controllo, e in caso di errore un crash di sistema o qualche altro errore fatale sarà inevitabile. Questa tecnica può essere usata solo con le procedure, e non con le funzioni.

CHR$

In molti casi la funzione CHR$ è usata per inserire un codice di controllo, ad esempio:

```
PRINT#1, CHR$(7);                    ' carattere (CTRL-G)
```

Quando l'argomento di CHR$ è una costante è sempre preferibile usare direttamente il carattere corrispondente; ad esempio la riga precedente può riscriversi:

```
PRINT #1, "simbolo 183 \f "Symbol" \s 10\symbol SIMBOLO \f "Symbol"";
```

dove il carattere tra le virgolette si ottiene premendo CTRL-P e poi CTRL-G, oppure premendo CTRL-P e poi componendo il numero 007 sul tasierino

numerico mentre si mantiene premuto il tasto ALT. Questo sistema può essere usato per tutti i caratteri di controllo ad eccezione dei seguenti:

ASCII 0	(non è possibile comporre la combinazione 000 sul tastierino)
ASCII 10	linefeed
ASCII 13	carriage return

Inoltre è preferibile non inserire come stringhe costanti i caratteri ASCII 1 e ASCII 2, che sono usate dal compilatore per altri scopi e tendono a confonderlo, come pure il carattere ASCII 26 (end-of-file). Questa tecnica è particolarmente vantaggiosa quando i caratteri di controllo appaiono all'interno di altre stringhe, in quanto permette di evitare il ricorso alla operazione di concatenazione

```
PRINT "Tre segnalazioni acustiche: uno... simbolo 183 \f "Symbol" \s 10\symbol SIMBOLO \f
"Symbol"due... simbolo 183 \f "Symbol" \s 10\symbol SIMBOLO \f "Symbol"...e tre !simbolo 183 \f
"Symbol" \s 10\symbol SIMBOLO \f "Symbol""
```

oppure quando è necessario inserire i caratteri di controllo nella dichiarazione di costanti simboliche.

```
CLOSE
```

Se si vuole risparmiare qualche byte, si tenga presente che non è strettamente necessario chiudere un file prima di aprire un nuovo file con lo stesso numero, in quanto il BASIC provvede a fare questo per noi. Inoltre, non è necessario chiudere i vari file usati dall'applicazione prima di ritornare al prompt di sistema, perché questa operazione è eseguita in ogni caso dal Dos .

```
CVI, CVL
```

Le funzioni CVI e CVL (e le loro simili CVS e CVD) erano originariamente usate nei programmi che gestivano file ad accesso random, per convertire il valore letto in una stringa in un INTEGER o un LONG. Anche se i programmi BASIC più recenti dovrebbero essere riscritti per utilizzare le variabili record (TYPE), più efficaci e strutturate, non di meno esistono numerose situazioni in cui CVI può fornire un aiuto prezioso. Ad esempio, per memorizzare i 16 bit meno significativi di un valore LONG in una variabile INTEGER si usa scrivere

```
value& = value& AND &HFFFF         ' elimina i bit più significativi
IF value&  <= 32767 THEN result% = value& ELSE result% = value& - 65536
```

con CVI la questione si risolve più elegantemente in questo modo

```
result% = CVI(MKL$(value&))
```

spiegazione: la funzione MKL$ restituisce una stringa di quattro caratteri corrispondente ai quattro byte che compongono il valore LONG passato come argomento; questi caratteri sono memorizzati secondo le solite convenzioni dei microprocessori Intel (la word meno significativa seguita dalla word più significativa), per cui CVI estrae proprio il valore memorizzato nella word meno significativa di value&.

Un problema (e una soluzione) simile si ha quando occorre estrarre il valore di una word memorizzata in una locazione di memoria; la soluzione classica è la seguente

```
result% = PEEK(offset%) + 256 * PEEK(offset% + 1)
```

che però provoca un overflow se il risultato è maggiore di 32767; occorre allora forzare il calcolo con valori LONG, in questo modo

```
value& = PEEK(offset%) + 256& * PEEK(offset% + 1)
```

ma si ripresenta il problema di convertire value& in un intero con segno; ecco invece la soluzione offerta da CVI

```
result% = CVI(CHR$(PEEK(offset%)) + CHR$(PEEK(offset% + 1)))
```

Un discorso simile può essere fatti per CVL; accade talvolta di voler ricomporre due INTEGER in un unico LONG (ad esempio, una indirizzo segmentato segmento:offset in un indirizzo a 32 bit). La soluzione in BASIC "puro" è molto complicata e poco efficiente, in quanto è necessario prevenire gli errori di overflow; è più immediato scrivere

```
result& = CVL(MKI$(lowWord%) + MKI$(highWord%))
```

Analogamente, per leggere una doppia word ad un indirizzo di memoria

```
result& = CVL(CHR$(PEEK(offset%)) + CHR$(PEEK(offset% + 1)) + _
        CHR$(PEEK(offset% + 2)) + CHR$(PEEK(offset% + 3)))
```

DATA

Non starò a spiegare come funziona l'istruzione DATA, ma mi limito ad un avvertimento: gli argomenti delle istruzioni DATA sono conservati nel codice del programma esattamente come sono scritte. Ad esempio, le istruzioni

```
READ a%, b%
DATA 12345,-22000
```

occupa esattamente 13 byte (11 byte per i dati più i separatori); questo è necessario in quanto il programma deve poter funzionare sia che il valore nelle DATA sia assegnato ad una variabile numerica sia che sia assegnato ad una variabile stringa. Anche senza conoscere nel dettaglio i meccanismi usati dal compilatore, è facile immaginare che le istruzioni

```
a% = 12345: b% = -22000
```

generino meno codice, in quanto le due costanti sono conservate nel file eseguibile come due word a 16 bit. Per questo motivo le istruzioni DATA dovrebbero essere usate con parsimonia.

DEF FN

L'istruzione DEF FN sembra essere abbastanza superata, rimpiazzata dalle più moderne e "strutturate" FUNCTION. In alcuni casi, tuttavia, questo costrutto può portare ad un notevole incremento della velocità di esecuzione. Provate ad esempio a cronometrare diecimila esecuzioni delle seguenti funzioni

```
FUNCTION Max% (x%, y%)
        IF x% > y% THEN Max% = x% ELSE Max% = y%
END FUNCTION

DEF FNMax% (x%, y%)
        IF x% > y% THEN FNMax% = x% ELSE FNMax% = y%
END DEF
```

E' facile vedere che **FNMax** è sensibilmente più veloce. La ragione è semplice: la FUNCTION deve poter essere richiamata da qualunque modulo del programma, per cui il compilatore crea le istruzioni Assembly relative ad una routine FAR, richiamando la routine con una istruzione CALL inter-segmento a 32 bit; nel caso del DEF FN (che non sono richiamabili dall'esterno del modulo in cui sono definite) la chiamata può essere effettuata con una CALL infra-segmento a 16 bit, per cui il compilatore produce codice più veloce e più compatto. Se però la dimensione del corpo della funzione aumenta, questo vantaggio iniziale si viene a perdere, per cui è meno vantaggioso definire funzioni complesse con DEF FN.

DEF SEG

Non è mai necessario modificare il valore di DEF SEG per accedere a dati che si trovano in DGROUP, ossia ai valori conservati nei vettori statici e nelle stringhe near. Ad esempio, se A$ è una stringa near, il codice seguente

```
DEF SEG = SSEG(a$)
acode% = PEEK(VARPTR(a$))
DEF SEG
```

può essere vantaggiosamente sostituito da

```
acode% = PEEK(VARPTR(a$))
```

in VBDOS *tutte le stringhe sono far, per cui la tecnica appena esposta non può funzionare correttamente.*

DIM

Le variabili dinamiche locali ad una SUB o FUNCTION dichiarate con DIM sono automaticamente azzerate ad ogni chiamata alla procedura stessa. Per questo motivo non è necessario azzerare esplicitamente le variabili numeriche e assegnare una stringa nulla alle variabili stringa

```
SUB MyProc (a, b, c)
DIM e%, f$
e% = 0:                  ' queste assegnazioni sono inutili
f$ = ""                  ' rallentano l'esecuzione
....
```

la stessa considerazione continua a valere se le variabili non sono esplicitamente dichiarate con DIM e la procedura o funzione non è qualificata con l'aggettivo STATIC

```
SUB MyProc (a, b, c)
e% = 0:                 ' queste assegnazioni sono inutili
f$ = ""                 ' rallentano l'esecuzione
....
```

E' buona norma, comunque, evitare le dichiarazioni implicite delle variabili dinamiche locali; può accadere infatti che, riprendendo a lavorare sul programma a distanza di tempo, ci si scordi di queste considerazioni e si aggiunga un aggettivo STATIC alla procedura, il che trasforma la variabile non dichiarata in una variabile statica e introduce alcuni bug molto difficili da trovare. Anche per questo motivo, il nuovo Visual BASIC per Dos include l'istruzione OPTION EXPLICIT, che rende obbligatoria la dichiarazione di tutte le variabili usate nel modulo in cui appare; si tratta di una opzione interessantissima, che dovrebbe essere sempre attivata nei proprio programmi.

Riguardo l'uso di DIM e REDIM per dichiarare array, si tenga presente che per dichiarare array più grandi di 64K è necessario lanciare l'interprete e compilare il sorgente con l'opzione **/ah**; se si tratta di un array di record, la dimensione di ogni singolo record dovrebbe essere una potenza del due (cioè 2, 4, 8, 16... byte), altrimenti non sarà possibile creare array più grandi di 128K anche ricorrendo all'opzione **/ah**.

In nessun caso è possibile creare array con più di 32767 elementi; fortunatamente questo limite è aggirabile abbastanza facilmente facendo uso di matrici multi-dimensionali. Ecco un esempio di codice che mostra come creare una matrice di 256 righe per 256 colonne in modo da simulare un array con 65536 elementi

```
' crea una matrice con 65536 elementi
' (il programma deve essere compilato con /ah)
DIM matrice!(255, 255)
' memorizza un valore nell'elemento (indice&)
matrice(indice& AND 255, indice& \ 256) = valore!
```

ENVIRON, ENVIRON$

Ecco due istruzioni BASIC decisamente poco sfruttate. Il problema di entrambe è che esse agiscono sulle variabili d'ambiente conservate nell'environment locale al programma BASIC, e non sul *master environment*. Al momento di lanciare un programma, il Dos crea una copia del master environment e ne passa l'indirizzo al programma; se evitiamo di modificare le variabili d'ambiente, questa "copia" dovrebbe essere identica all'originale, per cui la funzione

ENVIRON$ *dovrebbe* recuperare il valore delle variabili d'ambiente del master environment (il condizionale è necessario, in quanto è teoricamente possibile che nel frattempo un altro programma - ad es. un TSR - alteri il master environment). Si ricordi che ENVIRON$ richiede il nome della variabile in maiuscolo

```
comspec$ = ENVIRON$("COMSPEC")
nullstring$ = ENVIRON$("comspec")
```

in realtà è più corretto dire che ENVIRON$ è sensibile alla differenza tra le maiuscole e le minuscole,mentre l'istruzione SET del Dos converte automaticamente in maiuscolo il nome della variabile, per cui difficilmente troverete una variabile d'ambiente scritta in minuscolo. Se siete curiosi, provate a dare il comando SET senza argomenti in una finestra Dos da Windows 3.x: troverete una nuova variabile **windir** (in minuscolo), che prima non esisteva. Il nome di questa variabile è in minuscolo proprio per evitare che il suo contenuito sia cancellato accidentalmente durante la sessione Dos, ma essa può essere letta con una istruzione ENVIRON$ richiamata da un programma BASIC eseguito in una finestra Dos di Windows.

Anche il comando ENVIRON agisce esclusivamente sull'environment locale: tutte le modifiche fatte su questo environment sono scartate quando il programma termina, e in generale un programma BASIC non ha modo (a meno di manipolare direttamente la memoria, ma non è una tecnica semplice) di modificare il contenuto dell'environment principale; in altre parole. un programma può leggere il valore corrente di PATH, PROMPT e COMSPEC, ma non può modificarlo e fare in modo che queste modifiche persistano al termine dell'esecuzione. In pratica l'unico possibile uso di ENVIRON è quello di creare un environment particolare per un applicativo esterno lanciato con un comando SHELL. Ad esempio, un programma che prevede la possibilità di uscire temporaneamente al Dos potrebbe modificare il prompt per ricordare all'utente che sta lavorando con un processore secondario dei comandi, e quali sono i comandi da impartire per rientrare nel programma principale

```
ENVIRON "PROMPT=Digita EXIT+[invio] per rientrare nel programma $p$g"
```

Gli environment locali differiscono dal master environment per un particolare importante: le dimensioni del master enviroment sono impostate dall'opzione /E nella direttiva SHELL di CONFIG.SYS

```
SHELL=C:\DOS\COMMAND.COM /P /E:400
```

è possibile creare nuove variabili oppure modificare il valore assegnato a quelle esistenti, fintanto che la dimensione complessiva non supera il numero di byte indicati dall'opzione /E (400 nell'esempio precedente, oppure 160 se l'opzione /E è omessa). Viceversa, la dimensione dell'environment locale è determinata dalle variabili esistenti *al momento della sua creazione* - ossia il numero di byte effetivamente occupati nel master environment quando è

iniziata l'esecuzione del programma BASIC - ed in generale non è possibile creare nuove variabili né aumentare la lunghezza dei valori assegnati alle variabili esistenti. In altre parole, una istruzione di questo tipo

```
ENVIRON "PATH=" + ENVIRON$("PATH") + "C:\VBDOS\PROGRAMS"
```

è destinata a fallire provocando un errore di memoria insufficiente. Per fortuna esiste un rimedio abbastanza semplice, che consiste nel creare una o più variabili *dummy* ("fasulle") dal prompt del Dos con una istruzione SET prima di eseguire il programma

```
C:\VBDOS>  SET dummy=@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
```

Lo scopo del comando precedente è di aumentare artificialmente il numero di byte occupati nel master environment, che sarà "ereditato" dall'environment locale per il programma; a questo punto sarà sufficiente una istruzione

```
ENVIRON "DUMMY="
```

per liberare alcuni byte e avere un po' di spazio per creare o modificare le variabili esistenti o crearne di nuove.

ERR, ERL

Vi sono alcune caratteristiche che, pur documentate nei manuali del linguaggio, val la pena di sottolineare anche in questa sede. Il valore restituito dalla funzione ERR è automaticamente resettato a zero dalla istruzione RESUME o una delle sue varianti; dalle istruzioni ON ERROR e ON LOCAL ERROR e al termine di una SUB, FUNCTION o DEF FN.

Il comportamento di ERL presenta delle lievi differenze nei programmi interpretati e compilati; in caso di errore in un programma compilato, infatti, ERL restituisce un numero che indica l'ultima etichetta numerica incontrata nel corso dell'esecuzione; viceversa, nei programmi interpretati ERL restituisce un valore diverso da zero solo se è stata incontrata una etichetta numerica nella stessa procedura che ha causato l'errore.

EXP

Questa funzione restituisce un valore in singola precisione quando l'argomento è un intero oppure una espressione in singola precisione; in tutti gli altri casi restituisce un valore in precisione doppia. E' facile dimenticare questo dettaglio, ad esempio nel calcolo della costante neperiana e :

```
neper! = EXP(1)  ' risultato 2.718282
neper# = EXP(1#) ' risultato 2.71828182845905
```

FIELD

L'istruzione FIELD pone alcuni problemi, e dovrebbe essere evitata se possibile, eventualmente riscrivendo i programmi utilizzando le variabili strutturate (TYPE). Ad esempio, come avverte il manuale, non si dovrebbe utilizzare una stringa definita in una FIELD come argomento di un comando INPUT o nel membro sinistro di una operazione di assegnazione (usare *sempre* LSET e RSET).

Questi problemi derivano dal fatto che le stringhe menzionate in una istruzione FIELD sono caratterizzate da descrittore di stringa non standard, che punta al buffer del file anziché all'area di memoria normalmente associata alle stringhe. Una operazione di assegnazione o di INPUT, che modifica la dimensione della stringa, ricostruisce il descrittore "standard", che non punterà più al buffer. Per questo motivo tutte le successive operazioni di I/O non saranno corrette.

Uno dei pochi vantaggi di FIELD rispetto all'uso delle variabili record è di poter definire la struttura del database a runtime anziché al momento della compilazione; questa flessibilità permette, ad esempio, l'accesso a file di dati mediante tecniche di *data dictionary*, in cui la struttura dell'archivio è memorizzata su disco insieme ai dati stessi (nello stesso file o in file differenti).

FIX

Da tenere sempre a mente la differenza tra FIX e INT: il primo tronca un valore intero verso lo zero, mentre il secondo tronca sempre verso l'intero minore. Per argomenti positivi le funzioni forniscono sempre il medesimo risultato, mentre per valori negativi FIX tronca in effetti verso l'intero maggiore

```
PRINT FIX(-1.5)  ' restituisce -1
PRINT INT(-1.5)  ' restituisce -2
```

FRE

In BASIC PDS e Visual BASIC la funzione FRE(-3) restituisce la quantità di memoria espansa libera, che può essere usata dall'ambiente di sviluppo per

conservare parti di codice e (se è stato specificato lo switch /Ea) gli array numerici fino a 16K. La documentazione allegata al linguaggio non segnala, tuttavia, che se usata in programmi compilati questa funzione provoca un errore "Feature unavalaible" (codice 73). Pertanto, la forma FRE(-3) dovrebbe essere utilizzata esclusivamente nei programmi interpretati, e rimossa prima di compilare il programma.

FREEFILE

Se si intende scrivere delle procedure o funzioni che eseguono I/O su file, è buona norma evitare l'uso di numeri di file costanti nei comandi OPEN, PRINT, CLOSE, ecc. La funzione FREEFILE permette appunto di lasciare al BASIC il compito di calcolare il primo numero non ancora utilizzato, in questo modo:

```
filenum% = FREEFILE
OPEN "archivio.dat" FOR OUTPUT AS #filenum%
PRINT #filenum%, name$
...
CLOSE #filenum%
```

Il vantaggio di questo metodo è di ottenere delle procedure facilmente riutilizzabili in altri programmi e, con alcune versioni di BASIC, anche di ridurre leggermente le dimensioni del file eseguibile.

HEX$

Talvolta è necessario convertire un numero in una stringa esadecimale di lunghezza fissata; questo è il sistema più semplice

```
result$ = RIGHT$(STRING$(digits%, "0") + HEX$(number&), digits%)
```

dove evidentemente **number&** è il numero da convertire e **digits%** è il numero di cifre esadecimali desiderato. Questa seconda versione richiede qualche byte in meno

```
result$ = RIGHT$(STRING$(digits%, 48) + HEX$(number&), digits%)
```

se il valore di **digits%** è costante, il metodo si può semplificare ulteriormente; supponendo di voler ottenere un risultato con 4 cifre esadecimale, possiamo scrivere

```
result$ = RIGHT$("000" + HEX$(number&), 4)
```

Lo stesso criterio può essere applicato alla funzione OCT$.

IF...THEN...ELSE

Molti programmatori usano scrivere delle righe

```
IF opcode% THEN
```

in luogo di

```
IF opcode% <> 0 THEN
```

credendo in questo modo, di ottenere del codice più compatto e veloce; in realtà il compilatore BASIC produce il medesimo codice Assembly in entrambi i casi, per cui non si ottiene un vero e proprio risparmio nel codice eseguibile. La situazione è identica quando la variabile è di tipo SINGLE, DOUBLE o CURRENCY, mentre con le variabili di tipo LONG esiste in effetti un risparmio di due byte, mentre il guadagno in termini di velocità non è avvertibile a meno che l'istruzione in questione non si trovi in un ciclo eseguito molte migliaia di volte.

Quando il costrutto IF...THEN...ELSE serve per assegnare ad una variabile un valore piuttosto che un altro, come nel seguente frammento di programma:

```
IF LEN(a$) = 0 THEN opcode% = 0 ELSE opcode% = 20
```

si può riscrivere il codice in questo modo

```
opcode% = 20
IF LEN(a$) = 0 THEN opcode% = 0
```

ottenendo un risparmio di 16 byte, senza influenzare la velocità di esecuzione.

INKEY$

Dopo aver ottenuto una stringa corrispondente al tasto premuto dall'operatore si deve effettuare una serie di test per distinguere tra i vari tasti e comportarsi di conseguenza. Il metodo adottato più frequentemente è

```
k$ = INKEY$
IF k$ = CHR$(8) THEN
        ' tasto Backspace
ELSEIF k$ = CHR$(13) THEN
        ' tasto Invio
...
```

poichè i confronti sui valori INTEGER sono più efficienti di quelli che coinvolgono le stringhe, il codice precedente può essere ottimizzato in questo modo:

```
acode% = ASC(INKEY$)
IF acode% = 8 THEN
        ' tasto Backspace
ELSEIF acode = 13 THEN
        ' tasto Invio
...
```

Quando occorre testare i tasti del set ASCII esteso (tasti cursore, tasti funzione, combinazioni ALT+tasto, ecc.) si scrive comunemente un programma di questo tipo

```
k$ = INKEY$
IF k$ = CHR$(0) + CHR$(59) THEN
        ' tasto F1
ELSEIF k$ = CHR$(0) + CHR$(72) THEN
        ' tasto Freccia Su
...
```

Questo test può essere migliorato usando delle costanti stringa dove è possibile, ad es.

```
IF k$ = CHR$(0) + ";" THEN
```

dove evidentemente 59 è il codice ASCII del carattere ";". Ancora meglio è l'uso della funzione MKI$

```
IF k$ = MKI$(&H3B00) THEN
```

in questo caso è vantaggioso lavorare con le costanti esadecimali: la stringa restituita da MKI$ è lunga due caratteri, il primo dei quali corrisponde al byte meno significativo dell'argomento (zero nel nostro esempio), e il secondo corrisponde al byte più significativo (&H3B, che corrisponde in decimale al valore 59). Poiché abbiamo appena detto che le operazioni sui numeri interi sono sempre vantaggiose, possiamo fare a meno persino di MKI$ e lavorare invece con la funzione CVI

```
acode = CVI(INKEY$ + MKI$(0))
IF acode = &H3B00 THEN
```

notate l'operazione di concatenamento di stringhe nel primo rigo, necessaria per evitare un errore "Illegal Function Call" quando INKEY$ restituisce una stringa di uno o zero byte.

INPUT$

Sebbene sia normalmente utilizzata per la lettura da file, la funzione INPUT$ può essere utile per leggere i caratteri introdotti da tastiera. Ad esempio, ecco un metodo semplice per inserire una pausa nel programma

```
tmp$ = INPUT$(1)
```

Si noti che quando si legge la tastiera con INPUT$ i tasti del set ASCII esteso restituiscono un codice di un solo carattere, e non due come accade per la funzione INKEY$. Poiché INPUT$ non esegue l'eco su schermo dei caratteri introdotti, esso offre un metodo semplice ed efficace per l'introduzione di parole chiave

```
IF INPUT$(LEN(pass$)) <> pass$ THEN GOTO PasswordErrata
```

Vi è però una limitazione: non sono riconosciuti i tasti di editing (neanche BACKSPACE); se si preferisce forzare l'uso del tasto INVIO occorre modificare la riga precedente come segue:

```
IF INPUT$(LEN(pass$) + 1) <> pass$ + CHR$(13) THEN GOTO PasswordErrata
```

INSTR

Nell'usare questa funzione si tenga a mente che il confronto avviene considerando differenti le maiuscole e le minuscole; per forzare un confronto *case-unsensitive* occorre scrivere

```
result% = INSTR(start%, UCASE$(first$), UCASE$(second$))
```

Inoltre, se **second$** è una stringa nulla, la funzione restituisce sempre il valore di **start%**, oppure il valore 1 se il primo argomento numerico è omesso. INSTR può essere utile in numerose situazioni; ad esempio, ecco un programma che ricerca in un vettore la prima stringa che comincia con una sequenza di caratteri:

```
FOR i% = 1 TO UBOUND(array$)
        IF LEFT$(array$(i%), 4) = "abcd" THEN EXIT FOR
NEXT
```

INSTR ci offre una seconda possibilità

```
FOR i% = 1 TO UBOUND(array$)
        IF INSTR(array$(i%), "abcd") = 1 THEN EXIT FOR
NEXT
```

All'interno dell'interprete i due cicli hanno pressappoco lo stesso tempo di esecuzione, ma compilando il programma si può verificare che il secondo è circa il 30% più veloce ! Usata con accortezza, INSTR offre numerose possibilità;. consideriamo ad esempio la classica sequenza

```
' attendi una risposta S/N
DO
        k$ = UCASE$(INKEY$)
LOOP UNTIL k$ = "N" OR k$ = "S"
```

la precedente può essere semplificata in

```
DO
        k$ = INKEY$
LOOP UNTIL INSTR(" SsNn", k$) > 1
```

dove occorre tenere presente che, come ricordato in precedenza, se il secondo argomento è una stringa nulla INSTR restituisce 1. Anche quando le possibili scelte sono stringhe lunghe due o più caratteri, possiamo usare INSTR in luogo di una struttura SELECT CASE

```
' converti il nome del mese in un numero nell'intervallo 1-12
lista$ = ",gen,feb,mar,apr,mag,giu,lug,ago,set,ott,nov,dic,"
result% = (INSTR(lista$, "," + LCASE$(LEFT$(mese$, 3)) + ",") + 3) \ 4
IF result% = 0 THEN PRINT "Nome non valido"
```

È risaputo che se la ricerca della sottostringa fallisce, INSTR restituisce zero; in alcuni casi questo comportamento costringe a considerare dei casi particolari; nell'esempio che segue usiamo INSTR per trovare lo spazio che delimita la prima parola di una stringa, ma poiché la stringa **msg$** potrebbe non contenere alcuno spazio (cioé è composta di un'unica parola), siamo costretti a separare i due possibili casi con un costrutto IF..THEN..ELSE

```
' estrai la prima parola di una stringa
i% = INSTR(msg$, " ")
IF i% > 0 THEN word$ = LEFT$(msg$, i% - 1) ELSE word$ = msg$
```

Se invece abbiamo l'accortezza di aggiungere artificialmente uno spazio in coda alla stringa il codice ne risulta semplificato

```
word$ = LEFT$(msg$, INSTR(msg$ + " ", " ") - 1)
```

INTERRUPT, INTERRUPTX

C'è molto da dire su queste istruzioni, come avete modo di verificare in altre sezioni di questo stesso libro. In questa sede mi limito a considerare la differenza tra le due, che si limita al fatto che INTERRUPTX permette di impostare anche il valore dei registri di segmento DS e ES. Verrebbe da pensare che, limitandosi nel programma ad utilizzare una sola delle due si possa risparmiare qualcosa sulle dimensioni del file eseguibile, ma entrambe le routine sono state incluse nello stesso file OBJ, per cui richiamando una qualsiasi delle due l'altra viene automaticamente inclusa. Un piccolo guadagno di memoria è invece possibile per quanto riguarda le variabili TYPE usate per passare i valori ai/dai registri del microprocessore. Prima di tutto, come sottolinea anche il manuale del BASIC, la stessa variabile può essere usata sia per l'input che per l'output

```
DIM regs AS RegTypeX
...
INTERRUPTX &H21, regs, regs
```

Inoltre, e questo non è affatto menzionato nella documentazione, è possibile usare la struttura **RegTypeX** anche per il comando INTERRUPT, a condizione ovviamente di modificare in questo senso la dichiarazione nel file di include QB.BI / QBX.BI / VBDOS.BI (il nome varia a seconda della particolare versione del BASIC)

```
DECLARE SUB Interrupt (intnum%, inreg AS RegTypeX, outreg AS RegTypeX)
```

questo permette di risparmiare qualche byte nel segmento dati DGROUP, non essendo più necessario dichiarare due variabili distinte per INTERRUPT e INTERRUPTX.

LBOUND, UBOUND

Non c'è molto da dire su queste funzioni, se non che - quando vengono usate per calcolare il numero complessivo di elementi contenuti in un array - occorre sempre tenere presente l'elemento "di partenza"

```
numEls% = UBOUND(array) - LBOUND(array) + 1
```

Si ricordi anche che, se l'argomento è una matrice, queste funzioni restituiscono il numero di elementi nell'indice più a destra:

```
DIM matrix(2 TO 3, 5 TO 6)
PRINT LBOUND(matrix)          ' stampa 5
PRINT UBOUND(matrix)          ' stampa 6
```

Per ottenere informazioni su uno degli altri indici, occorre specificare un secondo argomento

```
PRINT LBOUND(matrix, 1)       ' stampa 2
PRINT UBOUND(matrix, 1)       ' stampa 3
```

LEN

In questo libro troverete che la funzione LEN è spesso usata nel test per una stringa nulla

```
' le due istruzioni seguenti sono equivalenti
IF a$ = "" THEN GOTO StringaNulla
IF LEN(a$) = 0 THEN GOTO StringaNulla
```

La seconda forma è più veloce, in quanto evita il confronto tra stringhe e riduce il problema al test di un valore INTEGER. Oltre al suo uso consueto per ricavare la lunghezza di una stringa, LEN può essere usata per determinare il numero di byte richiesti da una variabile semplice o strutturata

```
PRINT LEN(temp&)        ' stampa 4
PRINT LEN(num#)         ' stampa 8
TYPE myRecord
        nome AS STRING * 40
        codice AS LONG
END TYPE
DIM myRec AS myRecord
PRINT LEN(myRec)        ' stampa 44
```

Se usata in questo modo, il calcolo della funzione LEN viene effettuato dal compilatore e non rallenta il programma durantel'esecuzione.

LOCATE

LOCATE è una istruzione tuttofare, in grado di spostare il cursore in qualunque posizione dello schermo, controllare il suo stato attivo/disattivo e determina-

re il suo aspetto: questa versatilità si ottiene grazia alla possibilità di usare da uno a cinque argomenti. Poiché però la routine nella libreria runtime del BASIC prevede sempre e comunque 5 argomenti, il compilatore è costretto ad usare un complicato sistema per indicare quanti e quali argomenti sono effettivamente presenti. Questo comporta la generazione di più codice compilato di quanto sarebbe realmente necessario; per la precisione l'istruzione

```
LOCATE 2, 4
```

genera 22 byte invece dei 13 normalmente richiesti da una chiamata con due argomenti, mentre

```
LOCATE 2, 4, 1, 6, 7
```

richiede ben 43 byte, laddove per una chiamata con cinque argomenti sono normalmente sufficienti 25 byte. Per limitare questo spreco è possibile costruire alcune procedure ad hoc

```
SUB Locate2 (riga%, colonna%)
        LOCATE riga%, colonna%
END SUB

SUB CursorOn
        LOCATE , , 1
END SUB

SUB CursorOff
        LOCATE , , 0
END SUB
SUB CursorSize (curStart%, curStop%)
        LOCATE , , , curStart%, curStop%
END SUB
```

Richiamando queste procedure in luogo della LOCATE si otterrà una riduzione delle dimensione del codice, tanto più consistente quanto frequenti sono le istruzioni LOCATE rimpiazzate; lo svantaggio evidente di questo metodo è di rallentare leggermente la velocità di esecuzione. Le procedure di questo tipo si chiamano *wrapper procedure* (da wrap = avvolgere, in quanto racchiudono la procedura che intendono sostituire).

LOCK, UNLOCK

Queste istruzioni sono fondamentali nella preparazione di programmi per la rete locale; è importante ricordare che queste istruzioni devono essere sempre usate in coppia, e che ad ogni LOCK corrisponda in seguito una UNLOCK con gli stessi argomenti. Normalmente tutti i programmi dovrebbero essere "protetti" dagli errori fatali; questa precauzione è ancora più importante se il programma fa uso di questi comandi, in quanto un errore fatale che avviene dopo una LOCK e prima della UNLOCK corrispondente pone l'intero sistema in uno stato instabile. E' importante quindi l'uso accorto di una istruzione ON ERROR.

LOG

Occorre tenere a mente che - al pari di altre funzioni quali EXP e le funzioni trigonometriche ATN, COS, SIN e TAN - anche questa funzione restituisce un valore in singola precisione se l'argomento è un intero o in singola precisione, e restituisce un valore in doppia precisione negli altri casi. Ad esempio:

```
PRINT LOG(2)      ' risultato .6931472
PRINT LOG(2#)     ' risultato .693147180559945
```

LOG restituisce il logaritmo naturale del suo argomento, per calcolare il logaritmo in una base qualsiasi si può sfruttare la seguente formula:

```
LOG(n) in base K = LOG(n) / LOG(k)
```

il logaritmo decimale di un numero potrà allora essere calcolato in questo modo:

```
PRINT LOG(number#) / LOG(10#)              ' in doppia precisione
```

poiché il logaritmo è una funzione relativamente lenta, soprattutto in assenza del coprocessore matematico, è preferibile calcolare il valore del divisore una volta per tutte ed usare una costante

```
PRINT LOG(number#) / 2.30258509299405#
```

LPOS

Questa funzione - che restituisce il numero di caratteri inviati alla stampante dopo l'ultimo a-capo - è praticamente inutile, in quanto non tiene conto della espansione dei caratteri di tabulazione e di altri caratteri di controllo e non può essere usata per conoscere la posizione reale della testina di stampa. Se il vostro programma ha davvero questa necessità, è necessario gestire una variabile contatore incrementata per ogni carattere inviato alla stampante e azzerata per ogni a-capo.

LPRINT

LPRINT invia il suo output esclusivamente alla stampante LPT1; per sfruttare un'altra stampante eventualmente collegata al sistema occorre ricorrere ad una istruzione OPEN

```
OPEN "LPT2" FOR OUTPUT AS #1
PRINT #1, messaggio$
```

Allo stesso modo è possibile aprire LPT1, ma in tal caso è importante evitare di utilizzare LPRINT, in quanto i risultati sono imprevedibili. Un altro problema

minore è che, allo scopo di mantenere la compatibilità con il GW-BASIC, l'istruzione LPRINT CHR$(13) invia in effetti una coppia di caratteri CHR$(13) + CHR$(10), per cui è necessario che la stampante sia impostata correttamente.

LSET, RSET

Sebbene l'uso principale di questi comandi sia di memorizzare dei valori nelle stringhe definite in una istruzione FIELD (in preparazione per una operazione di PUT su file random), niente impedisce di usare anche con le stringhe normali, per allineare dei valori rispettivamente a destra o a sinistra di una stringa

```
' converti N% in una stringa di 10 caratteri, allineata a destra
result$ = SPACE$(10)
RSET result$ = RTRIM$(STR$(N%))
```

dove la funzione RTRIM$ è necessaria per eliminare lo spazio che STR$ aggiunge in coda alle cifre del numero. Per allineare a sinistra il metodo è simile

```
result$ = SPACE$(10)
LSET result$ = LTRIM$(STR$(N%))
```

In alcuni casi LSET può essere usata invece di LEFT$

```
result$ = LEFT$(a$, 4)
result$ = SPACE$(4): LSET result$ = a$
```

le due istruzioni precedenti sono equivalenti ed eseguono più o meno nello stesso tempo; se però sappiamo che la stringa **result$** di destinazione è già lunga il numero atteso di byte (quattro nel nostro esempio), la seconda riga può essere semplificata in

```
LSET result$ = a$
```

ottenendo un'incremento di velocità di circa il 20%; questo incremento è anche maggiore quando abbiamo la necessità di aggiungere un numero di spazi a sinistra di una stringa per portarla ad una determinata lunghezza. Ecco tre modi equivalenti per portare una stringa alla lunghezza di 40 caratteri

```
' metodo 1
result$ = LEFT$(a$ + SPACE$(40), 40)
' metodo 2: più veloce ma provoca errore se LEN(a$) > 40
result$ = a$ + SPACE$(40 - LEN(a$))
' metodo 3: il più veloce dei tre
result$ = SPACE$(40)
LSET result$ = a$
```

dove, come in precedenza, la prima istruzione nel terzo metodo può essere evitata se la stringa **result$** è già lunga 40 caratteri. Ovviamente quanto detto finora è valido anche per RSET nel caso in cui si desideri aggiungere degli spazi

all'inizio di una stringa. Oltre a poter agire sulle stringhe, LSET (ma non RSET) supporta una seconda sintassi

```
LSET recordVar1 = recordVar2
```

dove entrambe le variabili TYPE possono essere dichiarate con struttura e persino con lunghezza differente. Quando è utilizzata in questo modo, l'istruzione LSET non fa che copiare byte per byte il contenuto della variabile a destra del simbolo di uguale nella variabile che segue immediatamente il comando, senza curarsi del tipo delle componenti del record e badando unicamente a non copiare più byte del necessario (in altri termini, il numero dei byte copiati è dato dal minimo delle due lunghezze). Questo permette di trasferire valori tra due record di tipo differente senza dover eseguire una serie di assegnazioni sulle singole componenti, purchè i valori da trasferire occupino posizioni consecutive nel record destinazione a partire dalla prima

```
TYPE impiegato
        nome AS STRING * 30
        indirizzo AS STRING * 40
        citta AS STRING * 20
        qualifica AS STRING * 18
        codice AS INTEGER
END TYPE
TYPE anagrafica
        nome AS STRING * 30
        indirizzo AS STRING * 40
        citta AS STRING * 20
END TYPE
DIM impieg AS impiegato, anagr AS anagrafica
anagr.nome = "Rossi Mario"
anagr.indirizzo = "Via del Babbuino, 15"
anagr.citta = "Roma"
' assegna il contenuto di ANAGR alle prime componenti di IMPIEG
LSET impieg = anagr
```

Oltre a questo utilizzo abbastanza "ortodosso", il medesimo meccanismo permette una serie di "trucchi" davvero interessanti; ad esempio, è possibile suddividere un intero LONG nelle due word che lo compongono, in questo modo

```
TYPE LongType
        number AS LONG
END TYPE
TYPE IntegerType
        LowWord AS INTEGER
        HighWord AS INTEGER
END TYPE
DIM lo AS LongType, in AS IntegerType
' converti il numero conservato in value&
lo.number = value&
LSET in = lo
PRINT in.LowWord, in.HighWord
```

la trasformazione inversa è altrettanto semplice

```
' compatta due word in un unico intero a 32 bit
in.LowWord = n1%
in.HighWord = n2%
LSET lo = in
PRINT lo.number
```

LTRIM$, RTRIM$

Spesso queste funzioni sono usate per verificare se una stringa è nulla oppure contiene soltanto spazi

```
IF LEN(LTRIM$(a$)) = 0 THEN PRINT "Stringa nulla"
```

Un piccolo problema si presenta con le stringe a lunghezza fissa non inizializzate o con gli elementi stringa di una struttura TYPE; queste stringhe infatti, pur essendo logicamente da considerare stringhe nulle, non sono inizializzate con spazi bensì con caratteri ASCII 0 (come del resto avviene per tutti gli altri elementi di un record non inizializzato); per questo motivo le funzioni LTRIM$ e RTRIM$ falliscono sulle stringhe di questo tipo. Ecco un esempio:

```
DIM nome AS STRING * 20
a$ = RTRIM$(nome)               ' NOME non è inizializzato
PRINT LEN(a$)             ' stampa 20, anziché zero
PRINT ASC(a$)             ' stampa il valore 0
nome = "Rossi"
a$ = RTRIM$(nome)               ' NOME è inizializzato
PRINT LEN(a$)             ' stampa 5, corretto !
```

Una possibile soluzione è di inizializzare correttamente tutte le stringhe di lunghezza fissa usate dal programma, compreso le stringhe che figurano come componenti dei record. Per i nostri fini è sufficiente assegnare un singolo spazio:

```
DIM nomi(100) AS STRING * 40
FOR i% = 1 TO 100: nomi(i%) = " ": NEXT
PRINT LEN(LTRIM$(nomi(1)))        ' stampa il valore zero
```

Una soluzione alternativa, ma meno efficiente, è di confrontare la stringa con una seconda stringa lunga un ugual numero di caratteri e contenente solo byte nulli

```
DIM nome AS STRING * 20
a$ = RTRIM$(nome)
IF a$ = STRING$(LEN(a$), 0) THEN PRINT "Stringa nulla"
```

MID$

MID$ può essere usato sia come funzione che come comando; questa seconda forma non è però molto usata, anche se spesso offre la possibilità di ottimizzare delle porzioni di codice. Consideriamo il seguente esempio:

```
' leggi la stringa presente nella prima riga del video
result$ = ""
FOR col% = 1 TO 80
        result$ = result$ + CHR$(SCREEN(1, col%))
NEXT
```

Il programma precedente, perfettamente funzionante, è però inefficiente in quanto comporta numerose chiamate al gestore della memoria del BASIC (la stringa **result$** viene continuamente modificata, ogni volta cambiandone le dimensioni); ecco come si può ottenere lo stesso risultato in modo più efficiente

```
result$ = SPACE$(80)
FOR col% = 1 TO 80
        MID$(result$, col%, 1) = CHR$(SCREEN(1, col%))
NEXT
```

In questo secondo esempio la lunghezza della stringa non viene modificata nel ciclo, riducendo le chiamate al gestore di memoria del BASIC e la necessità di eseguire la *garbage collection*. Ecco un altro esempio di uso intelligente del comando MID$

```
' converte il maiuscolo la prima lettera di una stringa
' soluzione n.1
word$ = UCASE$(LEFT$(word$, 1)) + MID$(word$, 2)
' soluzione n.2, più efficiente
MID$(word$, 1, 1) = UCASE$(LEFT$(word$, 1))
```

MKDIR

Questa istruzione è in tutto e per tutto simile all'omonimo comando del Dos; la sola differenza sta nella possibilità di includere nel percorso degli spazi, e il manuale giustamente avverte che le directory create in questo modo non saranno accessibili dal prompt del sistema operativo. Questo però ci offre un semplice ed efficace metodo per "proteggere" alcuni file della nostra applicazione, non tanto per celarne il contenuto (ormai anche gli utenti più sprovveduti sanno utilizzare programmi come *PcTools* o *Norton Commander*...) quanto per evitare cancellazioni accidentali. Un esempio:

```
' crea una directory inaccessibile dal prompt del Dos
MKDIR "C:\DATI $$$"
' rendi corrente la directory appena creata
CHDIR "C:\DATI $$$"
```

MKI$, MKL$

Alcune proprietà di queste funzioni sono state già descritte illustrando le funzioni CVI e INKEY$. In generale, MKI$e MKL$ possono tornare utili quando

occorre creare stringhe contenenti più caratteri non stampabili, come se fossero varianti di CHR$ in grado di generare due o quattro caratteri

```
' invia la stringa ASCII 13 + ASCII 10 alla stampante
LPRINT CHR$(13) + CHR$(10);
LPRINT MKI$(&HA0D);
```

i due metodi sono equivalenti, ma il secondo richiede nove byte anziché 23, ed inoltre crea una sola stringa temporanea anziché tre e riduce indirettamente il ricorso alla garbage collection; notate che in questi casi è consigliabile usare le costanti esadecimali: il codice del primo carattere deve apparire nel byte basso dell'argomento, il codice ASCII del secondo carattere nel byte alto. Anche la funzione MKL$ può essere usata in questo modo, anche se l'occasione si presenta più raramente

```
' stampa una tabulazione, la stringa "01" e un carattere di salto pagina
LPRINT CHR$(9) + "01" + CHR$(12);
LPRINT MKL$(&HC313009);
```

Anche con MKL$ occorre ricordare che i quattro byte che compongono l'argomento devono apparire nell'ordine inverso rispetto ai caratteri della stringa che si intende formare.

NAME

Questa istruzione permette anche di spostare uno o più file ad un'altra directory, esattamente come il comando MOVE introdotto con la versione 6 di Ms-Dos e con l'unica limitazione che la directory originaria e quella di destinazione devono trovarsi sullo stesso drive. Ad esempio:

```
' sposta il file LEGGIMI nella directory C:\BACKUP
NAME "leggimi" AS "c:\backup\leggimi"
```

la riga precedente provoca errore se la directory di destinazione contiene un file con lo stesso nome del file sorgente; l'operazione di spostamento è molto più veloce della combinazione dei comandi COPY + DEL, in quanto il file non è realmente spostato, ma sono modificati solo le informazioni nelle due sottodirectory coinvolte.

ON ERROR

Questa istruzione è una delle più potenti del BASIC e, allo stesso tempo, uno dei comandi da evitare a tutti i costi se possibile. Infatti, se da un lato ON ERROR permette di creare dei programmi "protetti" e "a prova di errore" - che non terminano solo perché l'operatore ha dimenticato di chiudere lo sportellino

del drive o di collegare la stampante - dall'altro anche una singola istruzione di questo tipo impone che il modulo sia compilato con l'opzione /E o /X, creando in questo modo eseguibili di maggiori dimensioni e sensibilmente più lenti. E' importante comprendere il funzionamento dell'intrappolamento degli errori da parte del BASIC, sia per ottimizzare le prestazioni del programma che per giustificare alcuni comportamenti a prima vista poco comprensibili.

In un programma contenente un gestore di errori e compilato con l'opzione / E o /X dopo ogni istruzione è inserito un controllo che verifica la condizione di errore e salta se necessario alla routine appropriata, la quale termina in genere con una istruzione RESUME (che riesegue l'istruzione che ha causato l'errore) oppure RESUME NEXT (che passa ad eseguire l'istruzione seguente a quella che ha causato l'errore); l'istruzione RESUME prevede anche la possibilità di indicare una etichetta a cui saltare, ma questa caratteristica è sfruttata di rado nei programmi. Le informazioni addizionali inserite dal compilatore permettono anche di tenere traccia della posizione dell'ultima istruzione eseguita, in modo da poter saltare all'indirizzo corretto al momento di eseguire una istruzione RESUME o RESUME NEXT.

Nei programmi multimodulo non è necessario compilare tutti i file che compongono l'applicazione con le opzioni /E o /X, e di fatti usando il comando *Make* EXE *File* dell'ambiente di sviluppo sono compilati in questo modo soltanto i moduli che contengono un gestore di errore. E' interessante però vedere cosa accade quando si verifica un errore in un modulo in cui non è attivo il controllo degli errori; consideriamo il seguente esempio:

```
'____________________
' modulo UNO.BAS, compilato con /X
'____________________
COMMON SHARED errorCode%
ON ERROR GOTO ErrorHandler
CALL PrintFile "autoexec.bat"
PRINT "Fine Programma": END
ErrorHandler:
        PRINT "Errore #"; ERR
        errorCode% = ERR
        RESUME NEXT
'____________________
' modulo DUE.BAS, compilato senza /X
'____________________
SUB SomeFileOperation (filename$)
        OPEN filename$ FOR INPUT AS #1
        DO UNTIL EOF(1)
                LINE INPUT #1, i$
                IF errorCode% THEN EXIT DO
                PRINT #1, i$
        LOOP
        CLOSE #1
END SUB
```

Se avviene un errore nel modulo DUE.BAS, ad esempio il file da visualizzare non è stato trovato, il controllo passa alla correttamente etichetta **ErrorHandler**

in UNO.BAS, ma quando la routine per il trattamento degli errori esegue RESUME NEXT il programma riprende l'esecuzione alla istruzione che segue l'istruzione **CALL PrintFile**, perché è la più recente istruzione eseguita in UNO.BAS con il controllo degli errori attivato, e quindi la più recente istruzione per cui il BASIC ha avuto la possibilità di memorizzare l'indirizzo nel programma in esecuzione. Il programma precedente mostra un altro possibile errore dovuto al cattivo uso di ON ERROR: il gestore degli errori memorizza l'ultimo codice di errore nella variabile **errorCode%** condivisa da tutti i moduli del programma, in modo che la procedura **OpenFile** possa testare questa variabile per determinare se è avvenuto un errore in fase di lettura; evidentemente, per quanto appena detto, se l'istruzione LINE INPUT# provoca in errore il controllo non arriverà mai alla riga seguente, e il programma non avrà l'occasione di chiudere regolarmente il file aperto. La soluzione, ovviamente, è di compilare con /X tutti i moduli in cui può avvenire un errore.

Un uso molto poco ortodosso della istruzione ON ERROR, che certo non raccomando che può tornare utile in alcune particolari situazioni, consiste nello sfruttare a nostro vantaggio il sistema di intrappolamento degli errori del BASIC per permettere immediate uscite al programma principale dall'interno di chiamate nidificate a routine presenti in altri moduli. Supponiamo che il nostro programma preveda una routine fortemente ricorsiva, oppure due o più routine che si richiamano a vicenda:

```
' Programma principale
SHARED exitNow%
CALL RecursiveSub (20)
' Subroutine ----
SUB RecursiveSub (level%)
        IF INKEY$ = CHR$(27) THEN exitNow% = -1: EXIT SUB
        IF level% < 10 THEN
                PRINT "Entrata livello "; level%
                RicursiveSub level% + 1
                IF exitNow% THEN EXIT SUB
                PRINT "Uscita livello "; level%
        END IF
        IF INKEY$ = CHR$(27) THEN exitNow% = -1: EXIT SUB
END SUB
```

Se vogliamo aver la possibilità di uscire in qualunque momento premendo il tasto di ESCAPE siamo costretti ad inserire tante istruzioni **IF exitNow% THEN EXIT SUB** in modo da "disfare" tutte le chiamate in sospeso ed uscire "regolarmente" al programma chiamante. Grazie al costrutto ON ERROR e al comando ERROR possiamo semplificare il programma simulando un errore; per ottenere ciò dobbiamo però suddividere il programma in due moduli

```
'--------------------
' modulo MAIN.BAS, compilato con /X
'--------------------
ON ERROR GOTO ErrorHandler
CALL RecursiveSub (20)
PRINT "Fine Programma": END
```

```
ErrorHandler:
        RESUME NEXT
'___________________
' modulo RECURSE.BAS, compilato senza /X
'___________________
SUB RecursiveSub (level%)
        IF INKEY$ = CHR$(27) THEN ERROR 255
        IF level% < 10 THEN
                PRINT "Entrata livello "; level%
                RicursiveSub level% + 1
                PRINT "Uscita livello "; level%
        END IF
        IF INKEY$ = CHR$(27) THEN ERROR 255
END SUB
```

ON EVENT GOSUB

Al pari di ON ERROR, le numerose varianti di questa istruzione dovrebbero essere utilizzate solo se strettamente necessarie, poiché impongono una penalità al programma in termini sia di velocità di esecuzione che di dimensioni del codice eseguibile (e quindi di memoria occupata durante l'esecuzione). Le forme previste dal comando sono le seguenti

Istruzione	condizioni di salto alla subroutine
ON COM(n)	un carattere è ricevuto dalla porta seriale SERn
ON KEY(n)	è stato premuto il tasto specificato
ON PEN	è stata attivata la penna ottica
ON PLAY(n)	vi sono meno di *n* note nel buffer del comando PLAY
ON SIGNAL(n)	è stato ricevuto il segnale *n* in modo protetto (solo OS/2)
ON STRIG(n)	è stato attivato il joystick
ON TIMER(n)	sono trascorsi *n* secondi
ON UEVENT	è stato notificato un evento da una routine Assembly o C

Alcune di queste possono essere senz'altro trascurate in una discussione sul BASIC come linguaggio "moderno"; ad esempio, ON PEN è un retaggio delle prime versioni del GW-BASIC, quando la penna ottica godeva di una (limitatissima) diffusione. Analogamente, ON STRIG può interessare soltanto chi ha intenzione di scrivere videogames, attività per la quale il BASIC è certamente uno dei linguaggi meno indicati, soprattutto a causa delle scarse capacità grafiche; alla stessa categoria appartiene ON PLAY, che risulta molto poco versatile ai giorni nostri, abituati come siamo a suoni realistici prodotti da schede dedicate a 8 o 16 bit. Per motivi differenti può essere tralasciato anche

il comando ON SIGNAL, ormai solo un ricordo di quando la Microsoft aveva interessi nell'area OS/2. E' difficile giudicare l'utilità del comando ON UEVENT, in quanto è strettamente legata al compito della routine Assembly o C che genera il segnale. Restano quindi tre istruzioni soltanto: ON COM, ON KEY e ON TIMER. La loro convenienza deve essere valutata dal programmatore, soppesando da un lato l'*overhead* che esse impongono al programma e dall'altro l'efficacia e la facilità di un approccio differente, che faccia uso della tecnica del *polling*.

Il *polling* è un termine con cui si indica la tecnica di sondare continuamente la periferica in questione attendendo l'arrivo del segnale a cui siamo interessati; questa periferica può essere la tastiera o la porta seriale, ma può anche essere l'orologio di sistema. Nelle righe che seguono mostrerò come costruire un gestore "centralizzato" della tastiera che permette non solo di fare a meno della istruzione ON KEY, ma anche come sfruttare i tempi morti in cui il programma attende l'input dall'operatore:

```
FUNCTION GetInkey$ (ticks%)
        '____________________________
        ' Interroga la tastiera - sostituzione di INKEY$
        '   torna quando l'operatore preme un tasto oppure
        '   quando è trascorso il numero di ticks indicato
        '   se TICKS=-1 non tiene conto del timeout
        '____________________________
        DIM tickCount%, byte%, k$
        DEF SEG = 0
        tickCount% = ticks%
        byte% = PEEK(&H4C6)
        DO
                k$ = INKEY$
                IF PEEK(&H46C) <> byte% THEN
                        ' è trascorso un altro tick di sistema

                        byte% = PEEK(&H46C)
                        IF ticksCount% > 0 THEN ticksCount% = ticksCount% - 1
                END IF
                CALL BackgroundProcessing
        LOOP UNTIL LEN(k$) OR tickCount% = 0
        DEF SEG
        GetInkey$ = k$
END FUNCTION
```

La routine precedente dovrebbe essere usata in sostituzione di INKEY$, e permette di attendere la pressione di un tasto senza bloccare l'elaborazione del programma, anzi usando i tempi morti per processare dei dati in background; la subroutine **BackgroundProcessing** potrebbe ad esempio provvedere all'ordinamento o alla ricerca di dati in un archivio, ad eseguire il salvataggio dei dati ad intervalli prefissati, ad avvertire l'operatore di eventuali appuntamente o anche semplicemente per mostrare sullo schermo l'orario corrente. L'orizzonte, come al solito, è la fantasia del programmatore.

ON ... GOTO, ON ... GOSUB

Si tratta di due costrutti usati di rado, anche perché sono considerati "non strutturati" e associati ad un'epoca in cui il BASIC veniva usato per produrre il cosiddetto *spaghetti code*, i famigerati programmi in cui le terribili istruzioni GOTO si incrociavano con le etichette e i numeri di linea destinazione dei salti, in un groviglio che i colleghi d'oltreoceano amano associare al nostro piatto nazionale. Se mettiamo da parte i pregiudizi, queste due istruzioni possono essere usate per produrre un codice molto efficiente, più compatto e veloce di quello ottenibile con una struttura SELECT CASE o una cascata di IF...ELSEIF; l'unica condizione è che i valori da testare siano numeri interi e consecutivi:

```
PRINT "Introduci un numero 1-5"
:ripeti
        DO: n$ = INKEY$: LOOP UNTIL LEN(n$) = 1
        ON VAL(n$) GOTO sceltal, scelta2, scelta3, scelta4, scelta5
GOTO ripeti
```

questo programma funziona correttamente poiché se l'operatore introduce un valore nullo o maggiore di 5, oppure una stringa non numerica, l'istruzione ON GOTO è ignorata e il controllo passa alla istruzione seguente; l'unico difetto è che se l'operatore inserisce un valore numerico negativo o maggiore di 255 il programma si blocca con un errore "Illegal function call"; occorre allora aggiungere un controllo

```
PRINT "Introduci un numero 1-5"
:ripeti
DO: n$ = INKEY$: LOOP UNTIL LEN(n$) = 1
nv% = VAL(n$)
IF nv% < 1 OR n% > 5 THEN GOTO ripeti
ON nv% GOTO sceltal, scelta2, scelta3, scelta4, scelta5
```

Se i valori da testare non sono numerici o non sono consecutivi, ci viene in aiuto la funzione INSTR:

```
PRINT "(A)nnulla, (R)itenta, (I)gnora ?"
:ripeti
DO: k$ = INKEY$: LOOP UNTIL LEN(k$) = 1
ON INSTR("ARI", UCASE$(k$)) + 1 GOTO ripeti, annulla, ritenta, ignora
```

che funziona perfettamente poiché, se il tasto premuto non è uno di quelli ammessi, INSTR restituisce zero e l'istruzione ON GOTO provoca il salto alla etichetta **ripeti**.

OPEN

Questo comando supporta due sintassi:

```
OPEN file$ FOR mode [ACCESS access] [lock] AS #filenum [LEN length]
OPEN mode$, #filenum, file$ [,reclength]
```

dove la seconda è stata conservata quasi solo per ragioni di compatibilità con le vecchie versioni del linguaggio. In generale è sempre da preferirsi la prima forma, ma la seconda offre la possibilità di specificare il modo di accesso in una variabile stringa **mode$**, la qual cosa risulta utile in determinate situazioni. Ecco un esempio:

```
' apri un file in modo input, output o append
IF modo = 1 THEN
        OPEN file$ FOR INPUT AS #1
ELSEIF modo = 2
        OPEN file$ FOR OUTPUT AS #1
ELSE
        OPEN file$ FOR APPEND AS #1
END IF
' stesso effetto, usando la seconda sintassi
OPEN MID$("IOA", modo), #1, file$
```

è evidente che la seconda forma è decisamente meno leggibile dell'altra, ma in compenso è più compatta sia come sorgente che codice generato dal compilatore. Notate che non è possibile utilizzare questo metodo se occorre specificare la modalità di lock dei file.

Un'altro aspetto poco noto del comando OPEN è la possibilità di fare riferimento ai device driver supportati dal sistema operativo (CON, AUX, PRN, LPTn, ecc.). Ad esempio, è noto che il comando PRINT agisce sul canale standard per l'output, che in generale coincide con lo schermo ma che potrebbe essere ridirezionato su file o sul device NUL, il che è abbastanza probabile se il programma che state scrivendo è una utility; in questo caso, se volete mostrare una stringa di copyright o un messaggio di errore, e volete essere certi che questo appaia sullo schermo e non sia ridirezionato ad un file o a NUL, potete scrivere qualcosa del genere

```
OPEN "CON" FOR OUTPUT AS #16
PRINT #1, msg$
```

Se il vostro programma prevede la stampa su stampante e su file, potete creare due routine distinte, oppure semplicemente agire sul comando OPEN

```
IF stampaSuFile = 0 THEN
        OPEN "LPT1" FOR OUTPUT AS #1
ELSE
        OPEN filename$ FOR OUTPUT AS #1
END IF
PRINT #3, messaggio$
....
CLOSE #3
```

Tra l'altro, l'istruzione OPEN permette di accedere ad una qualsiasi delle tre porte parallele, e persino alle stampanti eventualmente collegate sulle porte seriali, mentre le istruzioni LPRINT possono solo fare riferimento alla prima stampante parallela:

```
' stampa sulla N-esima stampante parallela
OPEN "LPT" + LTRIM$(RTRIM$(STR$(n))) FOR OUTPUT AS #3
```

Nel riferirsi alle stampanti in questo modo si tenga presente che il device driver LPTn viene aperto in *character mode*, e che a tutti i carattere ASCII 13 (carriage return) è automaticamente accodato un carattere ASCII 10 (line feed); se però intendiamo usare la stampante in modo grafico, questa caratteristica interferisce con il flusso dei dati inviati al dispositivo, e deve essere disabilitata in questo modo

```
OPEN "LPT1:BIN" FOR OUTPUT AS #1
```

PCOPY

E' una istruzione sottoutilizzata, probabilmente perché è stata introdotta nelle versioni più recenti del linguaggio e alcuni "veterani" del BASIC non ne conoscono neanche l'esistenza. PCOPY permette di sfruttare parte della memoria addizionale presente sulle schede video CGA, EGA e VGA, per copiare il contenuto corrente dello schermo in una "pagina" in memoria o effettuare lo spostamento in direzione opposta: in questo modo è possibile, ed estramente facile, salvare e ripristinare schermate in modo testo:

```
' salva il contenuto corrente dello schermo
PCOPY 0, 1
...
' ripristina la videata originale
PCOPY 1, 0
```

Nell'usare questa istruzione occorre ricordare che nel modo a 25 righe x 80 colonne esistono quattro pagine disponibili, numerate da 0 (la pagina visibile) a 3; nei modi a 43 e 50 righe esistono due sole pagine, la pagina zero e la pagina uno, ma usando un numero di pagina maggiore per un uno dei due argomenti il BASIC non produrrà alcun messaggio di errore. Ricordate che i programmi che fanno uso di questa istruzione non saranno compatibili con le schede monocromatiche; inoltre, quando si cede il controllo ad un programma esterno con un comando SHELL è buona norma prevedere la possibilità che tale programma "sporchi" le pagine testo nascoste.

PEEK

Nella programmazione BASIC tradizionale la funzione PEEK non è molto usata, poiché uno dei compiti di un linguaggio ad alto livello è proprio quello di evitare al programmatore di dover ragionare in termini di indirizzi di memoria, segmenti e offset, ecc. In realtà, come è evidente dagli argomenti trattati in questo libro, non sempre è possibile scrivere programmi efficienti

limitandosi alle istruzioni standard del linguaggio, e in tal caso la funzione PEEK (e la sua compagna POKE) risulteranno utili per diversi "trucchi" molto interessanti.

Spesso è necessario leggere dalla memoria un intero a 16 bit, per cui occorre scrivere una riga di questo tipo

```
result% = PEEK(offset%) + 256 * PEEK(offset% + 1)
```

il comando precedente provoca errore di overflow se il valore all'indirizzo (offset+1) è maggiore di 127. Ecco la soluzione corretta, presentata sotto forma di DEFFN

```
DEF FNPeekW% (offset%)
FNPeekW% = CVI(CHR$(PEEK(offset% + 1)) + CHR$(PEEK(offset%)))
END DEF
```

PLAY

Non uso molto questa istruzione nei miei programmi BASIC, sia perché aggiunge un notevole overhead ai programmi compilati (circa 12K) ma soprattutto perché i risultati non sono quasi mai soddisfacenti: tutti i migliori giochi in commercio - ma anche molti giochi shareware - sfruttano le schede dedicate come Sound Blaster, AdLib, ecc., e al loro confronto il comando PLAY può al massimo suscitare tenerezza; d'altra parte, per gli effetti speciali o per simulare il suono di una astronave che piomba nell'iperspazio (anche se non ho mai capito come fanno a propagarsi nel onde sonore nel vuoto cosmico!) è preferibile usare il comando SOUND.

Fatta questa premessa, devo però ammettere che esiste qualche applicazione del comando PLAY, ad esempio nei programmi didattici per bambini non troppo smaliziati. In questa sede mi limito ad illustrare un uso inconsueto del comando PLAY

```
' inserisci una pausa di mezzo secondo
PLAY "P4"
```

il tempo di default per PLAY è "T120", cioè in un minuto sono contenute 120 note o pause da un quarto, ciascuna della quali è lunga quindi esattamente mezzo secondo; usando tempi e pause differenti è possibile ottenere praticamente qualsiasi durata, ad esempio

```
' una pausa di un decimo di secondo
PLAY "T150P16"
```

dove "T150" imposta a 0.4 sec. la durata delle note o pause da un quarto, per cui "P16" dura esattamente un decimo di secondo; misurando il risultato con precisione si nota che la pausa effettiva è leggermente più lunga (0.103 secondi), in quanto occorre tenere conto dell'esecuzione dell'istruzione PLAY.

POKE

Per questo comando vale quanto detto a proposito della funzione PEEK; usata con accortezza, POKE può anche migliorare l'efficienza delle operazioni sulle stringhe

```
' memorizza uno spazio nel primo carattere di una stringa
MID$(a$, 1, 1) = " "
' usare la seguente istruzione DEF SEG solo per le stringhe far
DEF SEG = SSEG(a$)
POKE SADD(a$), 32
```

la seconda soluzione è il 20% più veloce dell'altra, o anche più se **a$** è una stringa near in DGROUP e si rende quindi superfluo il comando DEF SEG.

PRINT

Non ci dovrebbe essere molto da dire sull'istruzione più usata del linguaggio BASIC. A molti però sfuggono alcune caratteristiche meno appariscenti, che invece vale la pena di ricordare; ad esempio, PRINT manda sempre il cursore a capo prima di stampare una stringa che verrebbe spezzata sulla 80a colonna del video. Per convincervene, provate ad eseguire il seguente breve programma

```
FOR i = 1 TO 100
        PRINT STRING$(RND * 32, "*");
NEXT
```

ci attenderemmo di ritrovare lo schermo completamente pieno di asterischi, ma in pratica il margine destro non è quasi mai rispettato, poiché quando la stringa è troppo lunga per la riga corrente il BASIC manda a capo il cursore. Ecco il modo più semplice per ovviare al problema

```
FOR i = 1 TO 100
        a$ = STRING$(RND * 32, "*")
        chars = 80 - POS(0)
        PRINT LEFT$(a$, chars); MID$(a$, chars + 1);
NEXT
```

A volte è necessario visualizzare sullo schermo caratteri con codice ASCII minore di 32, ad esempio per visualizzare il contenuto di un file binario oppure di un'area di memoria. Il problema è che il comando PRINT interpreta alcuni di quei caratteri come codici di controllo, ad esempio CHR$(12) riporta il cursore nell'angolo in alto a sinistra, CHR$(13) manda il cursore a capo, e così via. L'unica soluzione per questo problema consiste nel scrivere una procedura in grado di "scavalcare" il BASIC e scrivere direttamente nella memoria video

```
SUB VideoPrint (msg$)
        DEF SEG = &HB800                        ' scehda colore
                                                ' usare B000 per monocromatica
        offset = ((CSRLIN - 1) * 160 + POS(0) - 1) * 2
        FOR i = 1 TO LEN(msg$)
                POKE offset, ASC(MID$(msg$, i, 1))
                offset = offset + 2
        NEXT
        DEF SEG
END SUB
```

La procedura **VideoPrint** scrive alle coordinate correnti, ma per semplicità NON sposta automaticamente il cursore alla fine della stringa o al rigo seguente; inoltre i colori presenti sulle locazioni di schermo interessate non sono modificati. Sempre a proposito della istruzione PRINT, spesso è necessario stampare diverse stringhe o variabili stringa una di seguito all'altra; in questo caso le due soluzioni seguenti sembrano essere equivalenti

```
PRINT a$ + b$ + c$
PRINT a$; b$; c$
```

ma la seconda genera meno codice ed è più veloce; inoltre, poiché nel primo caso il BASIC deve generare una stringa temporanea (il risultato della concatenazione) e poi cancellarla in seguito, la prima soluzione tende a creare frammentazione nell'area delle stringhe e rende più frequente il ricorso alla *garbage collection*.

REDIM PRESERVE

Questa istruzione è stata introdotta nel BASIC PDS 7.1 e mantenuta nel Visual BASIC per Dos, per cui è possibile che sia passata quasi inosservata agli utenti delle vecchie versioni che hanno effettuato l'upgrade senza leggere attentamente il manuale del linguaggio.

Usando una versione precedente del linguaggio, quando occorre riempire un vettore con dei valori, ma non si conosce in anticipo il numero degli elementi richiesti, ci si ritrova con due possibili scelte: creare il vettore con maggior numero di elementi possibile oppure partire creando un numero "ragionevole" di elementi, aumentando la dimensione dell'array solo quando e se necessario. Questa seconda tecnica è ovviamente la più efficiente in termine di utilizzo della memoria, ma richiede uno sforzo supplementare per copiare gli elementi in un vettore temporaneo ed evitare che siano azzerati dal comando REDIM

```
' ridimensiona il vettore vet%() a 10000 elementi
REDIM temp%(LBOUND(vet%) TO UBOUND(vet%)
FOR index% = LBOUND(temp%) TO UBOUND(temp%)
        temp%(index%) = vet%(index%)
NEXT
REDIM vet%(10000)
```

```
FOR index% = LBOUND(temp%) TO UBOUND(temp%)
        vet%(index%) = temp%(index%)
NEXT
REDIM temp%(0)
```

Con REDIM PRESERVE è tutto più semplice, e non si corre il rischio di incorrere in un errore di memoria insufficiente al memento di creare l'array temporaneo:

```
REDIM PRESERVE vet%(10000)
```

RETURN

Pochi programmatori ricordano che RETURN supporta anche un numero di linea o il nome di una etichetta a cui saltare al termine di una subroutine:

```
RETURN etichetta
```

Questa è una delle istruzioni che ha fatto guadagnare al BASIC la fama di linguaggio "non strutturato" e di certo diminuisce la leggibilità del programma, eppure in alcuni casi si tratta di una possibilità che vale la pena sfruttare, soprattutto se corrisponde ad una situazione eccezionale e non al normale flusso di esecuzione del programma.

RND

La maggior parte di voi probabilmente sa perfettamente come ottenere un numero casuale in qualunque intervallo manipolando il risultato di RND, ma non costa niente ripeterlo una voltà di più:

```
casuale! = inizio! + RND * (fine! - inizio!)
```

se ci si limita ai numeri interi, la formula precedente deve essere ritoccata

```
casuale% = inizio% + RND * (fine% - inizio% + 1)
```

Qualunque programma che fa uso di questa funzione dovrebbe cominciare con una istruzione RANDOMIZE o RANDOMIZE TIMER per accertarsi che le sequenze generate ad ogni lancio del programma siano davvero casuali.

SADD, SSEG, SSEGADD

I valori restituiti dalle istruzioni SADD e SSEGADD possono variare nel corso dell'esecuzione del programma, per cui essi dovrebbero essere usati immediatamente e mai conservati in una variabile per utilizzarli in seguito. Una eccezione a questa regola si ha quando si è assolutamente certi che nel

frattempo non sia mai richiamata alcuna funzione del BASIC che modifica una stringa in memoria (anche se è una stringa differente da quella per cui abbiamo richiesto l'indirizzo) o ne crea una nuova (neanche una stringa temporanea). Ecco un esempio di tecnica "sicura"

```
SUB ReverseString (a$)
    '————————————————————
    ' inverti l'ordine dei caratteri in una stringa
    '————————————————————
    ' NOTA: in quickbasic 4.x la riga seguente può essere eliminata
    DEF SEG = SSEG(a$)
    length% = LEN(a$)
    inizio% = SADD(a$)
    fine% = SADD(a$) + length% - 1
    ' — sezione critica del codice, operazioni stringa non ammesse -
    FOR i% = 1 TO length% \ 2
            byte% = PEEK(inizio%)
            POKE inizio%, PEEK(fine%)
            POKE fine%, byte%
            inizio% = inizio% + 1
            fine% = fine% - 1
    NEXT
    ' — fine della sezione, le operazioni striga sono ammesse —
    DEF SEG
END SUB
```

Nell'accertarsi che durante la sezione critica non siano davvero eseguite operazioni su stringhe, si tenga anche presente eventuali istruzioni ON TIMER, ON KEY, ecc.; il tal caso potrebbe essere necessario racchiudere la sezione critica tra una coppia di istruzioni EVENT OFF e EVENT ON.

SCREEN

La funzione SCREEN è una delle più lente ed inefficienti del BASIC; ecco una prova

```
' leggi il contenuto della prima riga sul video
a$ = SPACE$(80)
FOR i% = 1 TO 80
        MID$(a$, i%, 1) = CHR$(SCREEN(1, i%))
NEXT
' stesso programma, con accesso diretto alla memoria video
a$ = SPACE$(80)
DEF SEG = &HB800
FOR i% = 1 TO 80
        MID$(a$, i%, 1) = CHR$(PEEK(i% + i% - 2))
NEXT
DEF SEG
```

È facile verificare che il secondo programma è quasi otto volte più veloce dell'altro. Questo guadagno è ancora più marcato quando si intende leggere più di un singolo rigo (o l'intera videata corrente), in quanto SCREEN richiede la scrittura di due cicli FOR nidificati, mentre PEEK permette di scorrere la memoria in modo lineare:

```
' leggi il contenuto corrente del video in una stringa
a$ = SPACE$(25 * 80)
DEF SEG = &HB800
FOR i% = 1 TO LEN(a$)
        MID$(a$, i%, 1) = CHR$(PEEK(i% + i% - 2))
NEXT
DEF SEG
```

Per quanto riguarda il *comando* SCREEN (da non confondere con la funzione omonima, di cui abbiamo appena discusso), non vi è molto da dire in quanto il manuale del linguaggio è abbastanza esauriente al riguardo (anzi, è l'istruzione meglio documentata, ben dodici pagine!). Vale però la pena di ricordare di usare, se possibile, una costante per specificare il modo grafico che si intende abilitare, e non una variabile: in questo modo il compilatore avrà la possibilità di scartare il supporto per tutti i modi grafici non necessari, e ottenere in questo modo una sensibile riduzione della dimensione del file eseguibile prodotto.

SGN

Questa funzione può rendersi utile quando occorre testare se due valori sono uguali e, in caso contrario, quale è il maggiore dei due; la soluzione "classica" si basa su una struttura IF..ELSEIF

```
IF n1 > n2 THEN
        GOTO maggiore
ELSEIF n1 < n2 THEN
        GOTO minore
ELSE
        GOTO uguale
END IF
```

Ecco la soluzione "alternativa", che sfrutta SGN e ON..GOTO e si riduce ad un'unica riga di codice

```
ON SGN(n1 - n2) + 2 GOTO minore, uguale, maggiore
```

STACK

Se il programma richiede molta memoria in DGROUP per le stringhe, le variabili semplici e gli array statici è possibile usare il comando STACK - presente soltanto in BASIC PDS e Visual BASIC per Dos - per diminuire il numero di byte assegnati normalmente allo stack

```
STACK 1024                  ' riserva un kbyte per lo stack
```

sapendo che la dimensione di default è di 3K, si vede che l'istruzione precedente rende liberi circa 2K in più per le variabili; la dimensione minima è di circa 350 byte.

In generale, diminuire la dimensione dello stack è una tecnica da evitare, in quanto una situazione di memoria insufficiente sullo stack può tradursi in un blocco totale del sistema; la dimensione minima "sicura" per lo stack dipende dal livello di nidificazione raggiunto nelle chiamate alle SUB e FUNCTION (e, in misura minore, GOSUB e DEF FN); se il programma include una procedura o funzione ricorsiva questa tecnica è fortemente sconsigliata, e spesso risulta necessario *aumentare* la dimensione dello stack per evitare errori fatali.

Anche nell'incrementare la dimensione dello stack vi sono dei limiti, dettati dall'occupazione corrente delle variabili in DGROUP; per allocare la maggior quantità possibile di memoria allo stack si deve eseguire il comando

```
STACK STACK
```

dove l'argomento del comando STACK è il valore restituito dalla funzione omonima.

STATIC

Questa parola chiave può essere usata sia nel dichiarare una variabile in una procedura (o funzione), sia nella dichiarazione della procedura stessa, come in

```
FUNCTION AreaQuadrato (x) STATIC
        AreaQuadrato = x * x
END FUNCTION
```

Quando è usata in questo modo, questa istruzione equivale a dichiarare STATIC tutte le variabili locali alla procedura. In generale le procedure e le funzioni STATIC sono leggermente più veloci e generano meno codice compilato, in quanto non sono necessarie le istruzioni per creare lo *stack frame* per le variabili locali e azzerare le variabili stesse in entrata alla procedura.

STRING$

Tra le due forme offerte da questa funzione:

```
' crea una stringa di asterischi
STRING$(length%, "*")
STRING$(length%, 42)
```

la seconda è senz'altro da preferirsi, in quanto è più veloce e non spreca spazio in DGROUP per la costante stringa. Dovendo creare una stringa di spazi, usare la funzione SPACE$.

SWAP

Questo comando del BASIC è spesso sottoutilizzato, quando al contrario permette spesso di velocizzare porzioni di codice; ad esempio, non molti sanno che l'operazione di assegnazione ad una variabile SINGLE o DOUBLE è almeno tre volte più lenta di un comando SWAP ! Ecco un caso in cui possiamo sfruttare a nostro vantaggio questa preziosa informazione

```
' assegna a A# il valore di B#, poi azzera B#
A# = B#: B# = 0
' stessa operazione, usando SWAP
A# = 0: SWAP A#, B#
```

è evidente il risparmio di tempo nel secondo caso, in cui possiamo ridurre ad una soltato le assegnazioni necessarie. Le stesse considerazioni sono valide per le operazioni sulle stringhe convenzionali (a lunghezza variabile), in cui per di più il guadagno cresce con la dimensione delle stringhe coinvolte nella operazione; cronometro alla mano, ho verificato che l'operazione di SWAP è circa quattro volte più veloce della assegnazione se agisce su stringhe di qualche decina di byte, e ben 15 volte più veloce su stringhe lunghe qualche centinaio di caratteri.

VAL

VAL restituisce sempre valori in virgola mobile, per cui è possibile introdurre inavvertitamente delle inefficienze in un programma. Se ad esempio dobbiamo ricercare un valore numerico in un array di stringhe, potremmo scrivere il seguente programma

```
FOR i% = 1 TO UBOUND(array$)
        IF VAL(array$(i%)) = number% THEN PRINT "Trovato": EXIT FOR
NEXT
```

questo metodo è poco efficiente, perché il confronto viene in realtà eseguito tra valori in virgola mobile; per quanto può sembrare strano - essendo ben nota la lentezza delle operazioni sulle stringhe - il programma

```
temp$ = LTRIM$(RTRIM$(STR$(number%)))
FOR i% = 1 TO UBOUND(array$)
        IF array$(i%) = temp$ THEN PRINT "Trovato": EXIT FOR
NEXT
```

è quasi CENTO volte più veloce del precedente ! Morale della storia: usate la funzione VAL con attenzione, e solo quando serve realmente. Altro in questo testo è riporatato il listato di una variante di VAL che restituisce valori interi.

VARSEG, VARPTR

Quando queste funzioni sono usate per ottenere l''indirizzo di un elemento di un array *dinamico*, occorre prendere alcune precauzioni in quanto il BASIC è in grado di spostare l'intero array nella memoria, ad esempio per creare spazio per nuovi vettori oppure in risposta ad una istruzione SETMEM o SHELL. In generale, quindi, il valore restituito da queste variabili dovrebbe essere utilizzato immediatamente e mai memorizzato in una variabile per usarlo più avanti nel programma. Il discorso è diverso se queste funzioni sono applicate ad oggetti memorizzati in DGROUP, e quindi statici; in tal caso è ragionevole supporre che tale valore non sarà modificato durante l'esecuzione del programma. Per quel che riguarda le stringhe convenzionali a lunghezza variabile, si fa spesso confuzione tra il significato del valore restituito da queste funzioni e quello restituito da SSEG / SADD:

- VARSEG e VARPTR restituiscono l'indirizzo del descrittore della stringa; poiché questo si trova sempre in DGROUP (anche se la stringa è del tipo *far*), il valore restituito da VARSEG coincide con quello restituito passando qualunque variabile semplce, e in generale non è necessario utilizzare DEF SEG per accedere al descrittore mediante PEEK e POKE
- SADD e SSEG restituiscono invece l'indirizzo della stringa vera e propria, ossia dei caratteri che la compongono; SSEG restituisce il valore corrente di DGROUP solo se la stringa è del tipo near, il che accade il QuickBASIC 4.x e nei programmi compilati con BC 7.x senza l'opzione /Fs; poiché in QuickBASIC tutte le stringhe sono near, la funzione SSEG non è disponibile.

WAIT

Si tratta di una istruzione poco usata, e tutto sommato superflua poiché essa può essere simulata con una combinazione di altre istruzioni

```
' le due righe seguenti sono equivalenti
WAIT portnum%, andExpr%, xorExpr%
DO: LOOP UNTIL (INP(portnum%) XOR xorExpr%) AND andExpr%
```

sebbene il comando WAIT sia certamente più compatto del ciclo DO...LOOP, il secondo ha il vantaggio di poter monitorare più porte di input o di eseguire nel frattempo qualche istruzione "utile" (il polling della tasiera), e soprattutto permette di evitare che il programma si blocchi in attesa di un segnale che non può arrivare, costringendo a resettare l'intero sistema

```
' stesso effetto di WAIT, ma esce quando l'operatore prema un tasto
DO: LOOP UNTIL ((INP(portnum%) XOR xorExpr%) AND andExpr%) OR LEN(INKEY$)
```

WIDTH

L'uso principale di questo comando è di impostare un numero differente di righe per l'output su video

```
WIDTH , 43              ' EGA o VGA, modo testo (SCREEN 0)
WIDTH , 50              ' VGA, modo testo (SCREEN 0)
WIDTH , 30              ' VGA, modo grafico (SCREEN 11-12)
WIDTH , 60              ' VGA, modo grafico (SCREEN 11-12)
```

La variante WIDTH LPRIINT è necessaria per inviare l'output ad una stampante a carrello largo (132 caratteri) oppure ad una stampante a 80 colonne ma che è impostata per lo stile compresso; in questo modo si evita che il BASIC aggiunga un carattere di a-capo (ASCII 13) una volta raggiunta la lunghezza della riga standard

```
WIDTH LPRINT 132 ' imposta la riga a 132 caratteri
WIDTH LPRINT 255 ' disattiva il controllo
```

2

LE FUNZIONI DOS

Uno dei luoghi comuni che affligge il povero BASIC è appunto di essere un linguaggio "povero", se confrontato al C e ad altri linguaggi più alla moda. In realtà i compilatori BASIC e C sono molto più simili di quanto non si creda, nel senso che entrambi sono in grado di trasformare una istruzione ad alto livello in una serie di istruzioni Assembly; il linguaggio C ha dalla sua parte la possibilità di eseguire delle ottimizzazioni molto spinte sulla velocità o compattezza del codice prodotto, ma nella maggior parte dei casi un programma BASIC scritto come si deve è quasi o altrettanto veloce di uno scritto in C, con l'enorme vantaggio di poter scrivere e testare il codice senza doverlo ricompilare ogni volta. Su un argomento, però, i detrattori del BASIC hanno sicuramente ragione: la libreria di funzioni del BASIC è in realtà molto scarna, e se ci limitasse alle sole istruzioni "normali" del BASIC sarebbero molti i programmi che non potrebbero essere scritti in questo linguaggio.

Oltre alla libreria runtime standard, che racchiude le istruzioni come PRINT, OPEN, ecc., il BASIC include una piccola libreria addizionale, contenente tre sole istruzioni: INTERRUPT, INTERRUPTX e ABSOLUTE. I manuali accennano a queste istruzioni quasi di sfuggita, ed è un peccato in quanto si può affermare che esse sono tra le più potenti e versatili istruzioni del linguaggio, vere miniere d'oro che nelle mani di un programmatore smaliziato permettono miracoli di ogni genere. A titolo di anticipazione di quello che presenteremo in questo capitolo e nel capitolo dedicato al BIOS, dirò che le istruzioni INTERRUPT e INTERRUPTX permettono di sostituire numerosi comandi BASIC standard come PRINT, LPRINT, WRITE, OPEN, CLOSE, SEEK, GET e

PUT (lettura/scrittura file), LOCK, UNLOCK, BSAVE, BLOAD, CHDIR, MKDIR, FILES, KILL, NAME, IOCTL, LOCATE, SCREEN, INPUT, PALETTE, nonché funzioni quali CSRLIN, POS, SEEK, LOC, LOF, EOF, DATE$, TIME$, INKEY$, INPUT$, FILEATTR, DIR$ e CURDIR$. Come vedete, un buon quarto delle parole chiave del linguaggio è in un certo senso superfluo, ma d'altra parte uno dei compiti di un linguaggio ad alto livello è proprio quello di assomigliare il più possibile alla lingua naturale.

Il vero interesse in queste istruzioni deriva però da tutte le possibilità che esse forniscono e che *non* sono ottenibili con il BASIC standard: la gestione del mouse, l'uso della memoria espansa, alcune funzioni avanzate delle schede grafiche EGA e VGA, operazioni più veloci con i file, la lettira dei parametri di configurazione del sistema, la memoria RAM installata, lo spazio libero su disco e innumerevoli altre piccole e grandi cose che possono dare il vero tocco di professionalità ai vostri programmi.

Come è possibile che due sole istruzioni siano così potenti ? In realtà, come vedremo, la potenza non è nelle istruzioni in quanto tali, ma nel fatto che esse danno la possibilità di richiamare un *interrupt software* del microprocessore. Senza spingerci troppo nei particolari, possiamo pensare ad un interrupt come ad una subroutine richiamabile da qualunque programma attivo in un particolare momento; esistono 256 interrupt distinti (numerati da zero a 255), ognuno dei quali corrisponde ad una categoria di funzioni. Per esempio l'interrupt 21h (la "h" che segue il numero indica che si tratta di un numero esadecimale, che in BASIC può anche scriversi &H21) permette di accedere a tutte le funzioni del sistema operativo, l'interrupt 33h permette di gestire un mouse Microsoft-compatibile, l'interrupt 10h accede alle funzioni video del BIOS, e così via.

UN PO' DI ASSEMBLY

E' un dato di fatto che tutti i programmatori un po' "smanettoni" dovrebbero per lo meno conoscere i rudimenti della architettura interna del microprocessore, dei registri, dei metodi di indirizzamento e del linguaggio Assembly in generale. Se poi intendete usare le istruzioni INTERRUPT e INTERRUPTX, non si tratta più di un semplice consiglio ma di una necessità, per cui è meglio chiarire subito le idee.

Come accade per una subroutine, un interrupt software può richiedere dei valori in ingresso e può restituire dei valori in uscita, ma a differenza delle procedure e funzioni scritte in BASIC, in Assembly i valori sono passati direttamente nei registri del microprocessore. Probabilmente molti di voi

sapranno che il microprocessore può conservare dei valori in alcuni *registri* a 16 bit, e in particolare:

```
il registro AX, detto anche "accumulatore"
i registri BX, CX e DX, detti "registri generali"
i registri DI e SI, detti "registri indice"
i registri ES e DS, detti "registri di segmento"
il registro dei flag
```

vi sono inoltre i registri SS, SP e BP, che servono per accedere ai dati sullo stack e non sono quasi mai manipolati direttamente dal programmatore, e i registri CS e IP che insieme puntano alla prossima istruzione da eseguire, e possono essere influenzati solo indirettamente per le istruzioni di salto (JMP) e di chiamata a subroutine (CALL).

I registri AX,BX,CX e DX presentano una caratteristica molto particolare: sebbene essi siano a tutti gli effetti dei registri a 16 bit, è possibile indirizzare anche solo i loro 8 bit inferiori oppure i loro 8 bit superiori. Ad esempio, l'accumulatore AX può essere visto come l'unione dei due registri AL e AH (dove L sta per *Low*, gli otto bit meno significativi, e H sta per *High*, gli otto bit più significativi); analogamente abbiamo BL-BH, CL-CH e DL-DH; gli altri registri sono invece sempre visti come 16 bit indivisibili. Il registro dei flag è particolare, in quanto esso non è destinato a memorizzare valori, ma i suoi singoli bit sono influenzati dal risultato delle operazioni del microprocessore (somme, confronti, ecc.); parleremo allora del *flag di zero* (che risulta attivato se il l'operazione precedente ha fornito un risultato nullo), del *flag di carry* (attivato se l'operazione ha generato un riporto), del *flag di segno* (influenzato dai risultati negativi), del *flag di overflow* (impostato dalle operazioni che provocano un overflow del risultato) e del *flag di parità* (attivato se il risultato ha un numero pari di bit non nulli).

Come abbiamo detto, ciascun interrupt software permette di accedere ad una categoria di funzioni, talvolta anche abbastanza vasta come nel caso dell'interrupt 21h che richiama tutti i possibili servizi del Dos. In tal caso si dice anche che il Dos installa un *interrupt handler*, ossia un "gestore di interrupt" che provvede a servire le richieste inviate dai programmi applicativi (tale handler risiede in memoria, ed il suo codice è caricato al bootstrap dal file di sistema MSDOS.SYS). Per richiedere un particolare servizio del Dos il programma deve inserire un valore nel registro AH prima di richiamare l'interrupt: tale valore specifica appunto quale *servizio* o *funzione* Dos intendiamo eseguire. Ad esempio, per richiedere il numero di versione Dos un programmatore Assembly scriverebbe

```
MOV   AH,30h   ; carica il valore 30h in AH
INT   21h      ; chiama l'interrupt del Dos
```

La funzione precedente restituisce il risultato nel registro AX, o più precisamente in AL il numero intero della versione, e in AH il numero decimale (ad

esempio il Dos 3.20 restituirebbe il valore 3 in AL, e il valore 20 in AH). Non tutte le funzioni del Dos sono così semplici e immediate; a volte, infatti, occorre fornire all'interrupt una stringa alfanumerica (ad es. il nome di un file o di una directory), che ovviamente non può essere memorizzata direttamente in un registro a 16 bit. In tal caso si usa indicare all'interrupt l'*indirizzo* della stringa, in una coppia di registri (un registro segmento più un altro registro usato per indicare l'offset). Ad esempio, il servizio 39h permette di creare una sottodirectory, il cui nome deve essere memorizzato all'indirizzo a cui punta la coppia di registri DS:DX. Il codice Assembly corrispondente sarà di questo tipo:

```
        MOV     DX,SEG nome       ; l'indirizzo segmentato della
        MOV     DS,DX             ;    stringa in DS:DX
        MOV     DX,OFFSET nome
        MOV     AH,39h            ; richiedi il servizio 30 hex
        INT     21h                     ; chiama l'interrupt del Dos
        ....
nome    DB      'miadir',0        ; la stringa, con un carattere
                                  ; ASCII 0 che ne indica il termine
```

GLI INTERRUPT IN BASIC

Torniamo a parlare di BASIC e vediamo la sintassi delle due istruzioni INTERRUPT e INTERRUPTX:

```
DECLARE Interrupt (intnum%, RegIn AS RegType, RegOut AS RegType)
DECLARE InterruptX (intnum%, RegIn AS RegTypeX, RegOut AS RegTypeX)
```

Come è stato detto in precedenza, una particolarità di queste istruzioni è di non appartenere al linguaggio BASIC vero e proprio, al pari ad esempio di PRINT o OPEN, e di essere invece procedure scritte in Assembly e contenute in una libreria separata. Chi lavora in QuickBASIC 4.0 o 4.5 le troverà nella Quick Library chiamata QB.QLB, chi usa il BASIC PDS 7.0/7.1 deve caricare la Quick Library QBX.QLB, e chi infine lavora con il Visual BASIC per Dos deve ricorrere a VBDOS.QLB. Al prompt del Dos daremo quindi il comando

```
QB /L QB              (in QuickBASIC)
QBX /L QBX            (in BASIC PDS)
VBDOS /L VBDOS        (in Visual BASIC per Dos)
```

Poiché in tutti i casi si tratta della Quick Library di default, il nome del file può essere omesso:

```
QB /L                 (in QuickBasic)
QBX /L                (in Basic PDS)
VBDOS /L              (in Visual BASIC per Dos)
```

Dopo aver caricato l'ambiente di sviluppo con la Quick Library adeguata, le due istruzioni non sono ancora disponibili, in quanto devono essere preventivamente "dichiarate" al compilatore, per indicargli quali argomenti esse si aspettano di ricevere, e di che tipo. Per questo motivo tra le prime righe

del programma occorre includere le due istruzioni DECLARE che abbiamo appena visto. Ma non basta: le due dichiarazioni fanno riferimento alle due strutture record RegType e RegTypeX, che devono pertanto essere dichiarate prima delle stesse DECLARE, in questo modo:

```
TYPE RegType              'per INTERRUPT
     ax    AS INTEGER
     bx    AS INTEGER
     cx    AS INTEGER
     dx    AS INTEGER
     bp    AS INTEGER
     si    AS INTEGER
     di    AS INTEGER
     flags AS INTEGER
END TYPE
TYPE RegTypeX             'per INTERRUPTX
     ax    AS INTEGER
     bx    AS INTEGER
     cx    AS INTEGER
     dx    AS INTEGER
     bp    AS INTEGER
     si    AS INTEGER
     di    AS INTEGER
     flags AS INTEGER
     ds    AS INTEGER
     es    AS INTEGER
END TYPE
```

Per fortuna non siamo costretti ogni volta a scrivere tutte queste istruzioni; la Microsoft ha pensato al nostro bene e ha preparato per noi in un file chiamato QB.BI, QBX.BI o VBDOS.BI (a seconda della particolare versione del linguaggio), quindi il tutto si riduce ad una istruzione di INCLUDE:

```
'$INCLUDE: 'qb.bi'          ; in QuickBasic
'$INCLUDE: 'qbx.bi'         ; in Basic PDS
'$INCLUDE: 'vbdos.bi'   ; in Visual Basic per Dos
```

A questo punto, dovrebbe essere abbastanza intuitivo capire come funzionano le istruzioni: (1) si dichiarano le due strutture RegIn e RegOut mediante una istruzione DIM (2) si caricano i valori in ingresso nei campi appropriati della struttura RegIn (3) si richiama l'interrupt con l'istruzione più appropriata, e infine (4) se si attende un risultato, lo si legge dai campi della struttura RegOut.

Qual è la l'istruzione più appropriata tra INTERRUPT e INTERRUPTX? Vedendo la dichiarazione delle due strutture RegType e RegTypeX, si può notare che esse differiscono esclusivamente per gli ultimi due campi, relativi ai registri di segmento ES e DS, quindi se la chiamata all'interrupt software non necessita di tali registri si può tranquillamente usare INTERRUPT, viceversa siamo obbligati ad usare INTERRUPTX. Se il vostro programma richiama alcuni interrupt che usano i registri ES e DS e altri che non li utilizzano, potete decidere di usare solo INTERRUPTX, poiché nel primo caso il valore presente negli ultimi due campi sarà automaticamente ignorato. Un'altro accorgimento interessante è quello di usare la stessa struttura di dati sia per i valori in ingresso che per i valori in uscita; oltre a ridurre il consumo di memoria questo

sistema è a mio parere decisamente più intuitivo di quello suggerito dal manuale: in definitiva, esiste un solo set di registri, non due. Ricapitolando, il nostro programma avrà una struttura di questo tipo:

```
'$INCLUDE: 'qbx.bi'            ; in Basic PDS
DIM reg AS RegTypeX            ; per l'istruzione INTERRUPTX
```

VERSIONE DOS E DRIVE CORRENTE

Siamo quasi pronti a chiamare il nostro interrupt, rimane solo un piccolo problema da risolvere: il BASIC vede soltanto numeri interi a 16 bit, quindi come posiamo caricare un valore in AH ? In realtà non è possibile caricare un valore in AH (o il AL, o in BL, ecc.) senza influenzare gli altri otto bit del registro, ma questo non è un problema, poiché il valore in ingresso in AL è ignorato dal Dos. Tutto diventa più chiaro lavorando in esadecimale: prendiamo come esempio un programma che deve conoscere la versione Dos sotto cui opera

```
'$INCLUDE: 'qbx.bi'            ; in Basic PDS
DIM reg AS RegTypeX            ; per l'istruzione INTERRUPTX
reg.ax = &H3000                ; carica 30h in AH, zero in AL
InterruptX &H21, reg, reg
```

in esadecimale infatti possiamo facilmente "vedere" i valori nelle due metà del registro AX; se avessimo usato la notazione decimale, la penultima riga sarebbe diventata

```
reg.ax = 122880
```

che non brilla certo per chiarezza. Una volta richiamato l'interrupt rimane solo da interpretare il risultato, restituito nei registri AL e AH; occorre quindi isolare il contenuto di AL e di AH, in questo modo:

```
al = reg.ax AND 255
ah = (reg.ax \ 256) AND 255
PRINT "Versione Dos "; al;"."; ah
```

Conoscere la versione del sistema operativo sotto cui opera un programma è spesso molto importante, perché il programma potrebbe fare uso di caratteristiche avanzate del Dos che sono state introdotte solo nelle versioni più recenti (ad es. il supporto per le reti locali, oppure l'uso della memoria superiore). A tale scopo ho costruito una piccola FUNCTION, che restituisce appunto il numero di versione Dos. Per poter effettuare confronti, la funzione restituisce un numero intero a tre cifre, del tipo XYY dove X è la parte intera e YY la parte decimale (ad esempio 320 per la versione 3.20). Ecco il listato:

```
FUNCTION DosVersion%
      DIM reg AS RegTypeX
      reg.ax = &H3000
      InterruptX &H21, reg, reg
      DosVersion% = (reg.ax AND 255) * 100 + ((reg.ax \ 256) AND 255)

END FUNCTION
```

Ad esempio, una delle prime istruzioni dei nostri programmi potrebbe essere:

```
IF DosVersion% < 300 THEN
      PRINT "Questo programma richiede Dos 3.00 o successivo"
      END
END IF
```

Se il programma contiene molte istruzioni INTERRUPT, può essere comodo dichiarare un'unica struttura **reg**, con visibilità "globale", invece di ricrearla in ciascuna funzione. In tal caso occorre porre tra le prime righe del programma l'istruzione:

```
      DIM SHARED reg AS RegTypeX
```

ed è possibile fare a meno dell'istruzione DIM nelle procedure e funzione che fanno uso della INTERRUPTX.

Un altro servizio Dos molto utile è la funzione 19h, che restituisce il nome del drive corrente; il codice del drive è restituito in AL, secondo il seguente schema: AL=0 per il drive A, AL=1 per il drive B, e così via. Anche in questo caso, possiamo scrivere una funzione autonoma:

```
FUNCTION CurrentDrive$
      DIM reg AS RegTypeX
      reg.ax = &H1900
      InterruptX &H21, reg, reg
      CurrentDrive$ = CHR$(65 + (reg.ax AND 255))
END FUNCTION
```

se il funzionamento della penultima riga non vi è chiaro, ricordatevi che la lettera "A" ha codice ASCII pari a 65, la lettera "B" è 66, ecc. Una volta definita la funzione possiamo scrivere:

```
      PRINT "Il drive corrente è "; CurrentDrive$
```

A chi lavora con il BASIC PDS 7.*x o* VBDOS *non sarà sfuggito il fatto che la funzione* CURDIR$ *restituisce nome della directory di lavoro, completo anche del nome del drive, e quindi l'istruzione precedente può scriversi (ma non in* Quick*BASIC 4.x) in questo modo:*

```
PRINT "Il drive corrente è "; LEFT$(CURDIR$, 1)
```

OPERAZIONI CON LE DATE

Un'altra funzione Dos che può essere facilmente inserita nei propri programmi è quella che restituisce la data corrente; si tratta quindi, almeno apparentemente, di un "doppione" della funzione DATE$, ma vedremo che ha qualcosa in più che la rende più interessante del previsto. Il numero del servizio è 2Ah, e restituisce nei registri CX, DH e DL rispettivamente l'anno, il mese ed il

giorno odierni. Poiché i valori restituiti sono più d'uno, una procedura è più adatta di una funzione

```
SUB Today (day%, month%, year%)
      DIM reg AS RegTypeX
      reg.ax = &H2A00
      InterruptX &H21, reg, reg
      day% = reg.dx AND 255
      month% = (reg.dx \ 256) AND 255
      year% = reg.cx
END FUNCTION
```

Come dicevamo, questo servizio ha un "plus" che lo rende interessante; a partire dalla versione 1.1 del Dos (e quindi, in pratica, in tutte le versioni del sistema operativo attualmente in circolazione), nel registro AL è anche restituito il giorno della settimana, 0 per domenica, 1 per lunedì, ecc. Possiamo perciò modificare la procedura precedente per aggiungere un quarto parametro, oppure creare una funzione ad hoc:

```
FUNCTION DayOfWeek%
      DIM reg AS RegTypeX
      reg.ax = &H2A00
      InterruptX &H21, reg, reg
      DayOfWeek% = reg.ax AND 255
END FUNCTION
```

Con un piccolo sforzo, possiamo anche ottenere il nome del giorno della settimana

```
FUNCTION NameOfDay$
      DIM reg AS RegTypeX
      reg.ax = &H2A00
      InterruptX &H21, reg, reg
      SELECT CASE reg.ax AND 255
              CASE 0: NameOfDay$ = "Domenica"
              CASE 1: NameOfDay$ = "Lunedì"
              CASE 2: NameOfDay$ = "Martedì"
              CASE 3: NameOfDay$ = "Mercoledì"
              CASE 4: NameOfDay$ = "Giovedì"
              CASE 5: NameOfDay$ = "Venerdì"
              CASE 6: NameOfDay$ = "Sabato"
      END SELECT
END FUNCTION
```

Il servizio "gemello" 2Bh *Set System Date* permette invece di impostare la data di sistema; si tratta quindi, almeno apparentemente, dello stesso effetto della seguente istruzione BASIC

```
              DATE$ = a$
```

con la differenza che il servizio Dos restituisce in AL un valore non nullo se la data non è corretta, e ci fornisce quindi un sistema per controllare la validità di una data introdotta dall'utente, senza essere costretti a considerare la lunghezza dei mesi, gli anni bisestili, e così via. Ecco una funzione che restituisce il valore -1 se una data è corretta, 0 in caso contrario:

```
FUNCTION IsDateValid% (giorno%, mese%, anno%)
      DIM reg AS RegTypeX, temp$
      ' memorizza la data di sistema
      temp$ = DATE$
      ' tenta di impostare la nuova data, usando
      ' il Dos; CX deve contenere l'anno, DH il
      ' mese e DL il giorno
      reg.cx = anno%
      reg.dx = (mese% * 256) + giorno%
      reg.ax = &H2B00
      InterruptX &H21, reg, reg
      ' se AL=0 la data è corretta
      IsDateValid% = ((reg.ax AND 255) = 0)
      ' ripristina la data odierna
      DATE$ = temp$
END FUNCTION
```

Come si vede, il formato dei registri è identico a quello per il servizio 2Ah mostrato in precedenza. Questa funzione mostra, tra l'altro, quanto può essere proficuo mischiare istruzioni Basic "normali" (la funzione DATE$) con le corrispondenti chiamate dirette al Dos.

INFORMAZIONI SU UN DISK DRIVE

I dati su disco sono organizzati in *settori*, mentre l'unità minima di allocazione dello spazio su disco sono i *cluster*: i settori sono quasi sempre lunghi 512 byte, mentre occorrono due, quattro, otto, sedici o più settori per formare un cluster (a seconda del tipo di disco), per cui un cluster può essere lungo da 1K a 8K. La memoria allocata a ciascun file è sempre un numero intero di cluster, per cui ad es. se un disco dispone di un cluster di 2K, un file lungo anche solo un byte impegna ben 2048 byte su disco, e analogamente un file di 2049 impegna due cluster (cioé 4096 byte), e così via. Provare per credere: eseguite un comando DIR e prendete nota dello spazio libero su disco, poi create un file di pochi byte, ad esempio con una istruzione

```
ECHO > file.tmp
```

a questo punto eseguite nuovamente il comando DIR e noterete che lo spazio libero si è ridotto non di pochi byte ma di 1K,2K,4K o 8K: questa è appunto la dimensione del cluster.

Questa introduzione era necessaria per comprendere appieno il valore del servizio 36h *Get Drive Allocation Information*, che richiede in DL il codice dei un drive (0=default, 1=A, 2=B, ecc.) e restituisce nei registri generali un numero di informazioni sul drive stesso, ed in particolare:

```
in AX il numero dei settori in ciascun cluster, o -1 in caso di errore
in BX il numero dei cluster liberi (non occupati dai file)
in CX il numero di byte in ciascun settore
in DX il numero complessivo di cluster nel disco
```

Poiché questo servizio restituisce più valori, è oppurtuno "incapsularlo" in una SUB piuttosto che usare una FUNCTION:

```
SUB DriveInfo (dr$, byPerSect%, sectPerClu%, totalClu%, freeClu%)
    '_______________________
    ' restituisce informazioni sul drive in DR$,
    ' oppure sul drive corrente se DR$ è una stringa nulla
    '_______________________
    DIM reg AS RegTypeX
    reg.dx = ASC(UCASE$(dr$) + "@") - 64
    reg.ax = &H3600
    InterruptX &H21, reg, reg
    byPerSect% = reg.cx
    sectPerClu% = reg.ax
    totalClu% = reg.dx
    freeClu% = reg.bx
END SUB
```

Notate il sistema usato per caricare in DL il codice del drive, oppure 0 se il parametro **dr$** è una stringa nulla; se non adottassimo il "trucco" di accodare un carattere "@", una eventuale stringa nulla causerebbe un errore "Illegal function call" al momento di eseguire la funzione ASC; sottraendo 64 al codice ASCII della stringa otteniamo 1 per il drive A, 2 per il drive B, ecc., e zero nel caso di una stringa nulla, valore che sta proprio ad indicare il drive di default.

Dai valori ottenuti è possibile calcolare altre quantità, che nella pratica sono persino più utili. In particolare, è possibile calcolare la dimensione di un cluster, la capacità del drive, e il numero dei byte liberi, in questo modo:

```
bytePerCluster% = sectperClu% * byPerSect%
totalBytes& = (totalClu% AND &HFFFF&) * sectPerClu% * byPerSect%
freeBytes& = (freeClu% AND &HFFFF&) * sectPerClu% * byPerSect%
```

Notate la trasformazione delle quantità **totalClu%** e **freeClu%** in interi LONG, che ha il compito di prevenire un overflow aritmetico, che avverrebbe certamente se il calcolo venisse effettuato esclusivamente con valori INTEGER a 16 bit. Gli ultimi due valori sono tanto importanti da meritare la scrittura di due apposite funzioni:

```
FUNCTION DiskSize& (dr$)
    '_______________________
    ' restituisce la capacità del disco in byte
    '_______________________
    DIM reg AS RegTypeX
    reg.dx = ASC(UCASE$(dr$) + "@") - 64
    reg.ax = &H3600
    InterruptX &H21, reg, reg
    DiskSize& = (reg.dx AND &HFFFF&) * reg.ax * reg.cx
END FUNCTION
FUNCTION DiskFree& (dr$)
    '_______________________
    ' restituisce lo spazio libero su un drive
```

```
    '___________________________
    DIM reg AS RegTypeX
    reg.dx = ASC(UCASE$(dr$) + "@") - 64
    reg.ax = &H3600
    InterruptX &H21, reg, reg
    DiskSize& = (reg.bx &HFFFF&) * reg.ax * reg.cx
END FUNCTION
```

Una ulteriore possibilità di questo servizio Dos è di permetterci di distinguere i vari tipi di disk drive, a partire dalla loro capacità. Ecco un frammento di programma che usa la funzione **DiskSize** appena definita:

```
LINE INPUT "Nome del drive: "; dr$
SELECT CASE DiskSize&(dr$)
        CASE 368640
                PRINT "Floppy disk 5.25 360K"
        CASE 737280
                PRINT "Floppy disk 3.5 720K"
        CASE 1228800
                PRINT "Floppy disk 5.25 1.2M"
        CASE 1474560
                PRINT "Floppy disk 3.5 1.44M"
        CASE ELSE
                PRINT "Hard disk"
END SELECT
```

per semplicità non sono prende in esame le possibilità di dischi virtuali ram o di altri device come CD-ROM o tape stream.

Se il drive indicato non esiste, il servizio 36h del Dos restituisce il valore -1 nel registro AX, e questa informazione ci offre lo spunto per scrivere una funzione che restituisce il nome dell'ultimo drive valido del sistema (che non è necessariamente il valore stabilito dalla direttiva LASTDRIVE in CONFIG.SYS). Il trucco è di partire dal drive Z e procedere verso le prime lettere dell'alfabeto, fermandosi quando si trova un drive per cui il servizio Dos restituisce in AX un valore differente da -1

```
FUNCTION LastValidDrive$
    '___________________________
    ' restituisce la lettera corrispondente
    ' all'ultimo drive valido nel sistema
    '___________________________
    DIM reg AS RegTypeX, index%
    FOR index% = 26 TO 1 STEP -1
            reg.dx = index%
            reg.ax = &H3600
            InterruptX &H21, reg, reg
            IF reg.ax <> -1 THEN
                    LastValidDrive$ = CHR$(index% + 64)
                    EXIT FUNCTION
            END IF
    NEXT
END FUNCTION
```

"INTERNAZIONALIZZIAMO" I NOSTRI PROGRAMMI

Avete mai avuto l'esigenza, o anche solo la voglia, di tradurre i vostri programmi in inglese, in francese o in qualche altra lingua ? Che ne direste di un programma che si adegua *automaticamente* alla lingua usata dall'utente ? Non occorre dotare il vostro software di doti medianiche e paranormali, cosa peraltro fuori dalla portata di questo libro, basta ancora una volta fare ricorso alla istruzione INTERRUPT e ai servizi del Dos.

Il sistema operativo MsDos può essere impostato per lavorare con tutte le lingue più diffuse, ad eccezione forse di qualche dialetto eschimese o polinesiano. La chiave di questa configurabilità è nel driver COUNTRY.SYS, ma per maggiori informazioni rimando ai manuali del sistema operativo. Quello che realmente conta per noi programmatori è che, a prescindere dal linguaggio usato dal Dos per i messaggi all'utente (che non è non modificabile), è sufficiente indicare con una direttiva COUNTRY un codice di nazionalità per fare in modo che il Dos utilizzi il formato giusto per le date, l'orario, i separatori dei decimali e delle migliaia, e così via.

Poiché è possibile interrogare il Dos e conoscere il codice della nazione impostata, nonché tutta una serie di altre utilissime informazioni, è anche possibile adattare i nostri programmi agli usi e costumi dell'utente senza neanche domandarglielo, purché il sistema operativo sia stato impostato con il codice corretto di nazionalità.

La chiave di tutto ciò è il servizio Dos 38h; a differenza di quelli visti in precedenza, il servizio in questione non restituisce le informazioni nei registri, in quanto il loro numero eccede certamente la disponibilità di registri dei processori 80x86; viceversa, esso usa un buffer di memoria lungo almeno 34 byte, il cui indirizzo deve essere passato alla routine nei registri DS:DX, ossia il segmento in DS e l'offset in DX. Poiché in BASIC non si possono allocare aree di memoria come in C, useremo l'accorgimento di creare una stringa di lunghezza fissa e di passare al servizio Dos l'indirizzo di questa stringa:

```
DIM reg AS RegTypeX, buffer AS STRING * 34
reg.ds = VARSEG(buffer)
reg.dx = VARPTR(buffer)
reg.ax = &H3800
InterruptX &H21, reg, reg
```

Il servizio 38h restituisce in BX il codice della nazionalità, ma attenzione: anche se non ci interessano gli altri valori restituiti nel buffer, siamo costretti a seguire la procedura indicata, in quanto il servizio Dos userebbe in ogni caso i valori presenti nei registri DS e DX e finirebbe per scrivere a casaccio in una zona del DGROUP, ossia del segmento che contiene tutte le variabili del BASIC, con le conseguenze sicuramente disastrose.

Prepariamo allora una funzione per determinare il codice di nazionalità del sistema su cui è eseguito il programma:

```
FUNCTION CountryCode%
      '_______________________
      ' restituisce il codice di nazionalità
      ' correntemente impostato per il Dos
      '_______________________
      DIM reg AS RegTypeX, buffer AS STRING * 34
      reg.ds = VARSEG(buffer)
      reg.dx = VARPTR(buffer)
      reg.ax = &H3800
      InterruptX &H21, reg, reg
      CountryCode% = reg.bx
END FUNCTION
```

Al ritorno dal servizio Dos, il buffer contiene numerose ed utili informazioni, nel seguente formato:

```
byte  significato
_________________________________________
01-02 formato della data (0=mm-gg-aa   1=gg-mm-aa   2=aa-mm-gg)
03-07 simbolo di valuta (formato ASCIIZ)
08-09 simbolo separatore delle migliaia (formato ASCIIZ)
10-11 simbolo separatore delle cifre decimali (formato ASCIIZ)
12-13 simbolo separatore per le date (formato ASCIIZ)
14-15 simbolo separatore per gli orari (formato ASCIIZ)
16    formato della valuta, codificato per bit
              bit 0   =0 se il simbolo precede il valore, 1 se lo segue
              bit 1   =0 se non vi è alcuno spazio tra simbolo e valore, 1 altrimenti
              bit 2   =1 se il simbolo di valuta sostituisce il separatore decimale
17    numero di cifre decimale nelle valute
18    formato dell'orario (0=12 ore  1=24 ore)
19-34 riservati o non importanti
```

Per estrarre i valori dal buffer, che nel nostro caso è una stringa a lunghezza fissa, dovremo usare la funzione MID$, eventualmente insieme alle funzioni CVI oppure ASC se si tratta di valori numerici. Ad esempio, per estrarre il formato della data si procede con:

```
formatoData = CVI(buffer)
```

mentre il formato dell'orario, che è memorizzato in un solo byte, si ottiene con

```
formatoOrario = ASC(MID$(buffer, 18))
```

Le stringhe ASCIIZ non sono altro che delle stringhe di caratteri la cui fine è marcata da un byte nullo, ossia ASCII 0 (la Z in ASCIIZ sta proprio per Zero); ad eccezione del simbolo di valuta, tutte le stringhe ASCIIZ riportate nel buffer sono lunghe due byte, e poiché il secondo è sicuramente ASCII 0, il primo è l'unico che realmente ci interessa.

La cosa più interessante è usare queste informazioni indirettamente, attraverso delle routine che formattano un valore passato dall'utente (una data, un orario, un numero o una valuta). Il seguente riporta la funzione **MyDate$**, che trasforma una data nel formato "MM-GG-AAAA" - il formato restituito dalla

funzione DATE$ del BASIC- nel formato atteso dall'utente. Si noti l'uso delle variabili statiche, che permettono di richiamare l'interrupt del Dos soltanto la prima volta ed accelerano quindi tutte le chiamate successive.

```
FUNCTION MyDate$ (dat$)
        '________________________________
        ' accetta una data in ingresso nel formato MM-GG-AAAA
        ' e la converte nel formato previsto dal Dos
        '________________________________
        STATIC alreadyCalled%, dateFormat%, sep$
        DIM reg AS RegTypeX
        IF alreadyCalled% = 0 THEN
                ' questo blocco è eseguito solo la prima volta
                ' che questa funzione è richiamata
                alreadyCalled% = -1
                reg.ds = VARSEG(buffer)
                reg.dx = VARPTR(buffer)
                reg.ax = &H3800
                InterruptX &H21, reg, reg
                dateFormat% = ASC(buffer)
                sep$ = MID$(buffer, 12, 1)
        END IF

        IF dateFormat% = 0 THEN
                ' formato americano
                MyDate$ = LEFT$(dat$, 2) + sep$ + MID$(dat$, 4, 2) + _
                        sep$ + RIGHT$(dat$, 4)
        ELSEIF dateFormat% = 1 THEN
                ' formato europeo
                MyDate$ = MID$(dat$, 4, 2) + sep$ + LEFT$(dat$, 2) + _
                        sep$ + RIGHT$(dat$, 4)
        ELSE
                ' formato giapponese
                MyDate$ = RIGHT$(dat$, 4) + sep$ + LEFT$(dat$, 2) + _
                        sep$ + MID$(dat$, 4, 2)
        END IF
END FUNCTION
```

Ecco invece la funzione **MyTime$**,che trasforma un orario nel formato "HH:MM:SS" (può essere il valore restituito dalla funzione TIME$) nella stringa equivalente nel formato previsto dal Dos, usando il giusto separatore ed aggiungendo i caratteri AM/PM per i paesi che usano il formato a 12 ore; si noti la complessa espressione necessaria per ottenere il giusto formato degli orari di tipo "PM".

```
FUNCTION MyTime$ (tim$)
        '________________________________
        ' accetta un orario in ingresso nel formato HH:MM:SS
        ' e lo converte nel formato previsto dal Dos
        '________________________________
        STATIC alreadyCalled%, timeFormat%, sep$
        DIM reg AS RegTypeX
        IF alreadyCalled% = 0 THEN
                ' questo blocco è eseguito solo la prima volta
                ' che questa funzione viene richimaata
                alreadyCalled% = -1
                reg.ds = VARSEG(buffer)
```

```
        reg.dx = VARPTR(buffer)
        reg.ax = &H3800
        InterruptX &H21, reg, reg
        sep$ = MID$(buffer, 14, 1)
        timeFormat% = ASC(MID$(buffer, 18))
    END IF

    IF timeFormat% = 1 THEN
        ' formato 24 ore, è sufficiente usare il giusto separatore
        MyTime$ = LEFT$(tim$, 2) + sep$ + MID$(tim$, 4, 2) + _
            sep$ + RIGHT$(tim$, 2)
    ELSEIF VAL(tim$) <= 12 THEN
        ' formato 12 ore, prima metà della giornata
        MyTime$ = LEFT$(tim$, 2) + sep$ + MID$(tim$, 4, 2) + _
            sep$ + RIGHT$(tim$, 2) + "AM"
    ELSE
        ' formato 12 ore, seconda metà della giornata
        ' questo caso è reso complicato dalla necessità di
        ' dover mantenere due cifre per l'orario, anche dopo
        ' aver sottratto 12
        MyTime$ = RIGHT$("0" + STR$(VAL(tim$) - 12), 2) + _
            sep$ + MID$(tim$, 4, 2) + sep$ + RIGHT$(tim$, 2) + "PM"
    END IF
END FUNCTION
```

Infine riporto il listato della funzione **MyNumber$**, che formatta un numero inserendo i caratteri di separazione di migliaia e dei decimali; anche in questo caso l'uso di variabili statiche ci permette di richiamare il servizio Dos soltanto alla prima esecuzione della routine. Notate come conviene trattare dapprima il numero come intero e positivo, e solo in seguito aggiungere eventualmente il segno meno e le cifre a destra del punto decimale

```
FUNCTION MyNumber$ (num#)
    '______________________________
    ' accetta un valore in ingresso, e lo formatta usando
    ' i separatori per migliaia e decimali previsti dal Dos
    '______________________________
    STATIC alreadyCalled%, tho$, dec$
    DIM reg AS RegTypeX
    DIM i%, tmp$
    IF alreadyCalled% = 0 THEN
        ' questo blocco è eseguito solo la prima volta
        ' che la funzione è richimata dal programma
        alreadyCalled% = -1
        reg.ds = VARSEG(buffer)
        reg.dx = VARPTR(buffer)
        reg.ax = &H3800
        InterruptX &H21, reg, reg
        tho$ = MID$(buffer, 8, 1)
        dec$ = MID$(buffer, 10, 1)
    END IF
    ' aggiungi i separatori per le migliaia
    tmp$ = LTRIM$(STR$(INT(ABS(num#))))
    FOR i% = LEN(tmp$) - 3 TO 1 STEP -3
        tmp$ = LEFT$(tmp$, i%) + tho$ + MID$(tmp$, i% + 1)
    NEXT
    IF number# < 0 THEN tmp$ = "-" + tmp$
    ' tratta separatamente il caso dei numeri non interi
    IF INT(num#) <> num# THEN
        tmp$ = tmp$ + dec$ + MID$(STR$(num#), INSTR(STR$(num#), ".") + 1)
```

```
        END IF
        MyNumber$ = tmp$
    END FUNCTION
```

A questo punto dovreste essere in grado di risolvere il problema, non affatto banale, di scrivere una funzione **MyCurrency$**, che formatta un valuta in ingresso usando il simbolo giusto e nella posizione giusta, i separatori per le migliaia e i decimali, ecc. Ricordate che il simbolo di valuta non è necessariamente lungo un solo carattere (per esempio, il simbolo delle lire italiane è "L.") e quindi deve essere estratto dal buffer cercando il byte nullo che termina la stringa ASCIIZ.

LE OPERAZIONI CON I FILE

Siamo arrivati finalmente alle operazioni con i file, uno dei cardini della maggior parte dei programmi applicativi: qualunque sia il campo in cui operiamo, prima o poi dovremo scrivere un programma che legge o salva dei dati su disco. Questo è uno dei settori in cui il BASIC non brilla particolarmente, non tanto per efficienza quanto per chiarezza. Ad esempio: un file può essere aperto in ben cinque modi differenti (INPUT, OUTPUT, APPEND, BINARY e RANDOM), ed alcune istruzioni di I/O valide in un modo non lo sono in un altro (i file aperti in modo OUTPUT supportano l'istruzione PRINT#, ma non la PUT#, i file aperti in modo INPUT supportano le istruzioni INPUT# e LINE INPUT# ma non la GET#, e così via); altre funzioni sono molto confuse, come la LOC che restituisce la posizione corrente all'interno del file se di tipo RANDOM, oppure il numero di record se di tipo BINARY, oppure ancora la distanza dall'inizio del file diviso 128 nel caso di file sequenziali. Per non parlare di tutte le altre opzioni del comando OPEN (ad es. l'opzione LEN) e del fatto che supporta due sintassi distinte.

Il motivo della presenza di tutte queste possibilità, spesso tra loro contrastanti, è sicuramente dovuto al fatto che il BASIC si è evoluto nel corso di oltre un decennio, ma ha mantenuto la compatibilità con *tutte* le versioni precedenti, al punto che è possibile compilare un programma scritto agli inizi degli anni Ottanta - magari progettato per un sistema non-MsDos - e trovare che gira perfettamente così com'è o che richiede solo qualche piccola modifica. In altri casi, il motivo dell'esistenza di alcune istruzioni apparentemente inutili è dovuto al tentativo di ottimizzare gli accessi al disco (è il caso, ad esempio, della opzione LEN del comando OPEN, che crea un sistema di bufferizzazione del file). Quasi sempre, però, le motivazioni "storiche" per la presenza di alcune istruzioni sono venute meno, e si può affermare che un buon numero di istruzioni per il trattamento dei file sono assolutamente superflue. Perso-

nalmente digerisco male tutte le limitazioni del linguaggio, e per le operazioni sui file tendo a "scavalcare" il BASIC e ad interagire direttamente con il Dos; quasi sempre i programmi ottenuti in questo modo sono più semplici e flessibili, spesso sono anche più efficienti.

Laddove il BASIC include cinque varianti per il comando OPEN, ognuna delle quali permette alcune operazioni ma ne inibisce altre, il Dos offre un unico servizio dell'interrupt 21h. In ingresso questo servizio prevede in AH il valore 3Dh (il numero della funzione), nei registri DS:DX il nome del file da aprire in formato ASCIIZ, e in AL un valore codificato a bit, come segue

```
bit 0-2 modo di accesso  000 = lettura
                             001 = scrittura
                             010 = lettura/scrittura
bit 3   riservato (zero)
bit 4-6 modo di condivisione   000 = compatibility mode
                             001 = impedisci qualsiasi operazione
                             010 = impedisci la scrittura
                             011 = impedisci la lettura
                             100 = tutte le operazioni sono permesse
```

E' evidente che manipolando questi bit è possibile sumulare tutte le varianti della OPEN, compreso le opzioni SHARED e LOCK, che hanno effetto solo se è stato caricato il modulo SHARE.EXE; l'unica condizione è che il file deve già esistere (in altre parole questa funzione non crea un file, ma apre un file esistente). Nel caso più semplice, AL contiene il valore 2 (file aperto in lettura/scrittura in compatibility mode); il seguente listato mostra la funzione **OpenFile**, che tenta di aprire un file e restituisce -1 in caso di errore, oppure l'handle del file se va tutto bene:

```
FUNCTION OpenFile% (nomefile$)
      '_______________________________
      ' Apri un file esistente
      ' restituisce l'handle del file, oppure un valore negativo
      ' in caso di errore
      '_______________________________
      DIM reg AS RegTypeX
      DIM buffer AS STRING * 80
      buffer = nomefile$ + CHR$(0)      ' crea una stringa ASCIIZ
      reg.ax = &H3D02                   ' apri un file in lettura/scrittura
      reg.ds = VARSEG(buffer)           ' in DS:DX l'indirizzo segmentato del
      reg.dx = VARPTR(buffer)           ' nome del file in formato ASCIIZ
      InterruptX &H21, reg, reg ' InterruptX perché usiamo DS
      IF reg.flags AND 1 THEN           ' testa il flag di carry
              OpenFile% = -1                    ' restituisce -1 in caso di errore
      ELSE
              OpenFile% = reg.ax                ' altrimenti restituisce l'handle
      ENDIF
END FUNCTION
```

Il concetto di *handle di file*, ben noto ai programmatori C, è simile al numero di file nel BASIC "standard", ad es. come il #2 in

```
OPEN "archivo.dat" FOR RANDOM AS #2
```

la differenza importante tra il numero di file e l'handle è che il primo è deciso dal programmatore, mentre il secondo è un valore a 16 bit restituito dal sistema operativo, ma che serve al medesimo scopo, cioè identificare univocamente il file per tutte le successive operazioni. In altre parole, dove in BASIC standard scriveremmo

```
ON ERROR GOTO ErroreFile
OPEN "archivo.dat" FOR RANDOM AS #2
```

per aprire un file bypassando il BASIC dovremmo scrivere:

```
handle% = OpenFile%("archivio.dat")
IF handle% = -1 THEN GOTO ErroreFile
```

Dov'è la convenienza ? Oltre a quanto detto prima a proposito di una maggiore flessibilità con le operazioni sui file, la convenienza è proprio nel fatto che non dobbiamo ricorrere alla istruzione ON ERROR, che come ben sappiamo rallenta sensibilmente l'esecuzione dell'intero programma. Esiste solo una seria limitazione: con questo metodo non è possibile intrappolare i cosiddetti errori critici del Dos, e cioè:

```
1. drive non pronto (il dischetto non è nel drive)
2. errore di disco o settore non trovato (supporto magnetico danneggiato)
3. disco protetto da scrittura
4. fine carta (ovviamente possibile solo per l'output su stampante)
```

In altre parole, se **OpenFile** fallisce per uno dei motivi sopra citati, il programma si interromperà col famigerato messaggio "Abort, Retry, Fail?"; viceversa, tutti gli altri errori sono invece riportati dalla funzione restituendo il valore -1 (poiché gli handle non possono essere negativi, siamo sicuri che nessuna operazione terminata con successo possa restituire quel valore). Sapendo che l'interrupt Dos restituisce in AX il codice di errore, i più volenterosi potrebbero modificare il listato precedente per far restituire da **OpenFile** un valore (negativo) che descriva il tipo di errore, in modo che il programma chiamante possa emettere un messaggio meno generico; l'unica piccola complicazione è che i codici di errore del Dos non corrispondono a quelli del BASIC, ed occorre preparare una piccola routine di conversione, usando come guida la seguente tabella, che però non è completa e non include tutti i possibili codici di errore restituiti dal Dos:

In realtà il Dos usa un sistema di numerazione di errori molto più sofisticato, in quanto prevede di ottenere un codice che specifica più in dettaglio in tipo di errore avvenuto, ed è anche in grado di offrire un consiglio sull'azione da intraprendere per ovviare all'errore stesso. I più curiosi dovrebbero procurarsi una *technical reference* del sistema operativo tra i sistemi consigliati in appendice.

Se siamo certi che il nostro programma non può incorrere in un errore critico, allora possiamo usare la **OpenFile**, altrimenti dobbiamo per forza ricorrere alla istruzione ON ERROR e alla OPEN tradizionale. Ma questo non significa

che dobbiamo rinunciare alle possibilità offerte dal Dos per il trattamento dei file. Infatti, il BASIC mette a disposizione una variante della funzione FILEATTR che restituisce appunto l'handle del file. Se ad esempio vogliamo aprire il file ARCHIVIO.DAT in lettura/scrittura approfittando dell'*error trapping* offerto dal BASIC, ed allo stesso tempo accedere al file senza limitazioni usando il Dos possiamo scrivere qualcosa del genere:

```
ON ERROR GOTO ErroreFile
OPEN "archivo.dat" FOR RANDOM AS #5
handle% = FILEATTR(#5, 2)
```

Non è necessario aprire il file in modo RANDOM, qualunque modo andrà bene ad eccezione di INPUT, che impedisce di leggere dati dal file. Se il file non esiste dobbiamo usare la funzione **CreateFile**, riportata qui di seguito, che come la precedente restituisce l'handle del file appena creato oppure un valore negativo per indicare errore

```
FUNCTION CreateFile% (nomefile$)
	'______________________________
	' Crea un file
	' restituisce l'handle del file, oppure un valore negativo
	' in caso di errore
	'______________________________
	DIM reg AS RegTypeX
	DIM buffer AS STRING * 80
	buffer = nomefile$ + CHR$(0)		' il nome del file in formato ASCIIZ
	reg.ax = &H3C00
	reg.cx = 32				' imposta l'attributo di archivio
	reg.ds = VARSEG(buffer)			' in DS:DX l'indirizzo segmentato del
	reg.dx = VARPTR(buffer)			' nome del file, in formato ASCIIZ
	InterruptX &H21, reg, reg ' InterruptX perché usiamo DS
	IF reg.flags AND 1 THEN			' testa il flag di carry
		CreateFile% = -1		' restituisce -1 in caso di errore
	ELSE
		CreateFile% = reg.ax		' altrimenti restituisce l'handle
	ENDIF
END FUNCTION
```

(in questo caso è generato errore anche se il file esiste già su disco).

Una volta aperto un file cone **OpenFile** o **CreateFile** possiamo leggere informazioni da esso, oppure scriverne: al contrario del BASIC che offre decine di varianti per l'I/O da file, il Dos ne permette due soltanto: la funzione 3FH per la lettura e la funzione 40H per la scrittura. I due servizi sono simili, ed entrambi richiedono in BX l'handle del file, in CX il numero dei byte da leggere o scrivere, in DS:DX l'indirizzo segmentato della zona di memoria da cui prelevare i byte (in caso di scrittura) oppure in cui memorizzare i byte (in lettura). Le seguenti routine fanno proprio questo, e sono concepite come funzioni perchè restituiscano un valore non nullo in caso di errore

```
FUNCTION ReadFile% (handle%, bytes%, segment%, offset%)
	'______________________________
	' legge.un numero di byte da un file in un buffer
	' restituisce un valore non nullo in caso di errore
```

```
	'______________________________
	DIM reg AS RegTypeX
	reg.ax = &H3F00              ' richiama il servizio 3F, Read File
	reg.bx = handle%             ' handle in BX
	reg.cx = bytes%              ' numero dei byte da leggere in CX
	reg.ds = segment%            ' indirizzo dei dati in DS:DX
	reg.dx = offset%
	InterruptX &H21, reg, reg' InterruptX perché usiamo DS

	' il Dos restituisce in AX il numero di byte effettivamente letti
	' ed il flag di carry settato se vi è un errore (non critico)
	' quindi occorre testare due condizioni per accertarsi che non
	' vi sia alcun errore e che tutti i dati nel buffer siano validi
	IF (reg.flags AND 1) OR reg.ax <> reg.cx THEN
		ReadFile = -1
	ELSE
		ReadFile = 0
	ENDIF
END FUNCTION


FUNCTION WriteFile% (handle%, bytes%, segment%, offset%)
	'______________________________
	' scrive un numero di byte su file da un buffer
	' restituisce un valore non nullo in caso di errore
	'______________________________
	DIM reg AS RegTypeX
	reg.ax = &H4000              ' richiama il servizio 40, Write File
	reg.bx = handle%             ' handle in BX
	reg.cx = bytes%              ' numero dei byte da scrivere in CX
	reg.ds = segment%            ' indirizzo dei dati in DS:DX
	reg.dx = offset%
	InterruptX &H21, reg, reg' InterruptX perché usiamo DS

	' il Dos restituisce in AX il numero di byte effettivamente scritti
	' ed il flag di carry settato se vi è un errore (non critico)
	' quindi occorre testare due condizioni per accertarsi che non
	' vi sia alcun errore e che tutti i dati nel buffer siano stati scritti
	IF (reg.flags AND 1) OR reg.ax <> reg.cx THEN
		WriteFile% = -1
	ELSE
		WriteFile% = 0
	ENDIF
END FUNCTION
```

Supponiamo di avere un file composto di record, la cui struttura è definita nel programma da una istruzione TYPE appropriata. Ecco allora come utilizzare la nuova funzione **ReadFile** per leggere un record dal file

```
DIM re AS RecordType
IF ReadFile(handle%, LEN(re), VARSEG(re), VARPTR(re)) THEN
	GOTO ErroreFile
END IF
```

Se sappiamo a priori quanti record contiene il file, ed è possibile caricare tutto il file in un array di record, possiamo scrivere un ciclo:

```
DIM ar(numrecord%) AS RecordType
FOR n% = 1 TO numrecord%
	res% = ReadFile(handle%, LEN(ar(n%)), VARSEG(ar(n%)), _
```

```
                VARPTR(ar(n%)))
            IF res% THEN GOTO ErroreFile
        END IF
```

Fin quì niente che non si possa fare con una istruzione GET, ma ora viene il bello: il servizio Dos permette di leggere fino a 65535 byte in un'unica operazione, quindi se l'array occupa meno di 64K è possibile leggere l'intero l'array di record in un'unica operazione, sfruttando il fatto che i record occupano locazioni contigue di memoria:

```
DIM ar(numrecord%) AS RecordType
temp& = LEN(ar(1)) * numrecords%
IF temp& <= 32767 THEN bytes% = temp& ELSE bytes% = 65536 - temp&
res% = ReadFile(handle%, bytes%, VARSEG(ar(1)), VARPTR(ar(1)))
IF res% THEN GOTO ErroreFile
```

A questo punto è necessaria una precisazione: il BASIC è in grado di trattare solo interi con segno, mentre il Dos attende nel registro CX un valore *unsigned* che può essere compreso tra 0 e 65535; fintanto che il valore da esprimere è minore o uguale a 32767 non ci sono problemi, ma per valori maggiori la moltiplicazione in seconda riga darebbe errore se eseguita su valori INTEGER; la IF in terza riga serve per calcolare il valore a 16 bit da passare alla routine **ReadFile**. E' superfluo sottolineare che la lettura di un intero vettore di elementi in questo modo è molto più veloce del loop sui singoli record.

Dovendo lavorare con interi senza segno, può convenire definire due funzioni che eseguano le conversioni oppurtune

```
FUNCTION Signed% (value&)
      ' Converte un valore unsigned a 16 bit in un INTEGER
      Signed% = CVI(MKL$(value&))
END FUNCTION

FUNCTION Unsigned& (value%)
      ' Converte un valore INTEGER in un valore unsigned a 16 bit
      Unsigned& = (value% AND &HFFFF&)
END FUNCTION
```

Usando queste funzioni, possiamo scrivere un frammento di programma che scriva su disco il vettore di record:

```
bytes% = Signed%(CLNG(LEN(ar(1)) * numrecords%)
res% = WriteFile(handle%, bytes%, VARSEG(ar(1)), VARPTR(ar(1)))
IF res% THEN GOTO ErroreFile
```

Le funzioni **ReadFile** e **WriteFile** non sono limitate all'I/O di record; qualunque zona di memoria contigua può essere letta e scritta in questo modo, ad es. una schermata 25x80 in modo testo:

```
res% = WriteFile(handle%, 4000, &HB000, 0)      'schede monocromatiche
```

oppure:

```
res% = WriteFile(handle%, 4000, &HB800, 0)      'schede a colori
```

Tutti gli array e le matrici possono essere lette e scritte con queste funzioni, ad eccezione degli array e matrici di stringhe convenzionali (a lunghezza variabile): infatti, in quest'ultimo caso, l'indirizzo restituito da VARSEG/

VARPTR si riferisce al descrittore della prima stringa dell'array, ma le stringhe vere e proprie sono memorizzate altrove e in generale non occupano indirizzi consecutivi nella memoria.

Dopo le operazioni sul file non rimane che chiuderlo. Se il file era stato aperto da BASIC con una OPEN non resta che usare una CLOSE corrispondente; se invece il file era stato aperto con **OpenFile** o **CreateFile**, bypassando del tutto il BASIC, è necessario usare la funzione **CloseFile**, che come al solito restituisce un valore non nullo in caso di errore

```
FUNCTION CloseFile(handle%)
        '______________________________
        ' Chiude un file aperto con OpenFile
        ' restituisce un valore non nullo in caso di errore
        '______________________________
        DIM reg AS RegType
        reg.ax = &H3E00
        reg.bx = handle%
        Interrupt &H21, reg, reg
        IF reg.flags AND 1 THEN
                CloseFile = -1
        ELSE
                CloseFile = 0
        ENDIF
END FUNCTION
```

Si ricordi che il file viene comunque chiuso automaticamente dal Dos quando il programma termina; questo però non è vero se il programma è eseguito in forma interpretata, ed infatti uno svantaggio di questa tecnica è di dover chiudere esplicitamente i file con delle istruzioni **FileClose**; se il programma interpretato termina per un errore inaspettato, è necessario procedere con la chiusura "manuale" dei file impartendo il comando **FileClose** dalla finestra *Immediate*. Se però il programma è già terminato e non si conosce il valore degli handle da usare nel comando, è preferibile salvare il programma, uscire dall'ambiente di sviluppo e rieseguirlo subito dopo; se non si segue questa procedura gli handle dei file rimasti aperti non saranno più recuperabili, e dopo un certo numero di esecuzioni "errate" può avvenire che gli handle a disposizione del programma si siano ridotti al punto tale da non poter più garantire una esecuzione corretta, e che addirittura non vi sia più la possibilità di salvare il programma su disco. In tal caso, si impartisca il seguente comando in modo immediato:

```
ercode% = FileClose(3)
```

in modo da liberare un handle ed essere quindi almeno in grado di salvare il programma.

IL PUNTATORE AL FILE

Per ogni file aperto, il Dos conserva il valore del puntatore ad esso, ossia la posizione nel file da cui saranno letti i prossimi dati, oppure a cui saranno scritti i prossimi valori. La funzione 42 hex del Dos permette di modificare il puntatore al file. Ecco i valori che il Dos attende nei registri:

AH = &H42

AL = metodo di posizionamento

0 = offset è assoluto (dall'inizio del file)

1 = offset è relativo alla posizione corrente

2 = offset è relativo dalla fine del file

BX = l'handle del file

CX:DX = il valore del nuovo offset (32 bit)

Quando AL=0 questa funzione è simile al comando SEEK del BASIC, con una importante differenza: per il Dos il primo byte del Dos è il byte zero, viceversa per il BASIC il primo byte è il byte uno. Se si tiene conto di questa differenza, è facile scrivere una procedura che sostituisca il comando SEEK per i file aperti bypassando il BASIC: notate il sistema usato per caricare l'offset a 32 bit nella coppia di registri CX e DX, senza provocare errori di overflow.

```
SUB FileSeek (handle%, offset&)
      '___________________________
      ' Posiziona il puntatore ad un file
      '___________________________
      DIM reg AS RegTypeX
      reg.ax = &H4200           ' usa il metodo AL = 0
      reg.bx = handle%
      reg.cx = CVI(RIGHT$(MKL$(offset&), 2))
      reg.dx = CVI(MKL$(offset&))
      InterruptX &H21, reg, reg
END SUB
```

La procedura non prevede il controllo degli errori: infatti l'unico caso in cui il serivzio 42h del Dos può provocare errore è quando l'handle passato dal programma è errato e non corrisponde ad alcun file aperto.

A differenza del comando SEEK, la funzione 42 del Dos permette anche di indicare l'offset relativo alla posizione corrente del file, oppure alla fine del file. La seguente procedura permette di saltare di un certo numero di byte in avanti o indietro, molto utile quando si lavora con singoli record; notate che se ci limitiamo a salti di max. 32767 byte la dimensione massima di un record) possiamo semplificare la funzione, in quanto i sedici bit più significativi dell'offset in CX saranno sempre zero (per i salti in avanti) oppure -1 (per i salti all'indietro).

```
SUB FileSkip (handle%, numBytes%)
      '————————————————————————
      ' Sposta il puntatore ad un file di un numero
      ' di byte in avanti o indietro (max 64K)
      '————————————————————————
      DIM reg AS RegTypeX
      reg.ax = &H4201            ' usa il metodo AL = 1
      reg.bx = handle%
      reg.cx = (numBytes% < 0)  ' imposta CX = 0 o -1
      reg.dx = numBytes%                  ' numero dei bytes in CX:DX
      InterruptX &H21, reg, reg
END SUB
```

Sfruttando le peculiarità del servizio 42h possiamo scrivere una procedura **FileSeekToEnd**, che sposta il puntatore alla fine del file, e la funzione **OpenFileForAppend** che usa lo stesso accorgimento per simulare il comando OPEN FOR APPEND:

```
SUB FileSeekToEnd (handle%)
      '————————————————————————
      ' sposta il puntatore alla fine del file
      '————————————————————————
      DIM reg AS RegTypeX
      reg.ax = &H4202            ' usa il metodo AL = 2
      reg.bx = handle%           ' (cx=0, dx=0)
      InterruptX &H21, reg, reg
END SUB

FUNCTION OpenFileAppend% (nomefile$)
      '————————————————————————
      ' Apri un file esistente in modo Append
      ' restituisce l'handle del file, oppure un valore negativo
      ' in caso di errore
      '————————————————————————
      DIM reg AS RegTypeX
      DIM buffer AS STRING * 80
      buffer = nomefile$ + CHR$(0)        ' crea una stringa ASCIIZ
      reg.ax = &H3D02                     ' apri un file in lettura/scrittura
      reg.ds = VARSEG(buffer)             ' in DS:DX l'indirizzo segmentato
      reg.dx = VARPTR(buffer)             ' del nome del file in formato ASCIIZ
      InterruptX &H21, reg, reg' InterruptX perché usiamo DS
      IF reg.flags AND 1 THEN             ' testa il flag di carry
            OpenFileAppend% = -1          ' restituisce -1 in caso di errore
      ELSE
            OpenFileAppend% = reg.ax ' altrimenti restituisce l'handle in AX
            reg.bx = reg.ax               ' handle in BX
            reg.ax = &H4202               ' il puntatore alla fine del file
            reg.cx = 0                    ' offset null relativo alla fine del file
            reg.bx = 0
            InterruptX &H21, reg, reg
      ENDIF
END FUNCTION
```

Un altra caratterestica interessante della funzione 42H è di restituire nella coppia dei registri CX:DX il nuovo valore del puntatore al file; il listato seguente mostra come sfruttare questo comportamento per creare l'equivalente della funzione SEEK del BASIC: il metodo seguito consiste nel richiedere di "spostare" di zero byte il puntatore rispetto alla posizione corrente, e leggere il "nuovo" valore del puntatore:

```
FUNCTION FileOffset& (handle%)
      '______________________________
      '  Restituisce il valore corrente del puntatore al file
      '______________________________
      DIM reg AS RegTypeX
      reg.ax = &H4201            ' usa il metodo AL = 1
      reg.bx = handle%           ' CX=0, DX=0
      InterruptX &H21, reg, reg
      FileOffset& = CVL(MKI$(reg.dx) + MKI$(reg.cx))
END FUNCTION
```

Sfruttando una tecnica molto simile è anche possibile determinare la lunghezza di un file aperto con una istruzione OPEN o con la funzione **OpenFile**: il trucco, in questo caso è di spostare temporaneamente il puntatore alla fine del file, prendendo quindi nota della sua nuova posizione e riportandolo al valore originario. Questo metodo è più affidabile rispetto a quello classico che si basa sulla lettura della directory: infatti, se il file è stato scritto ma non è ancora stato chiuso è possibile, anzi molto probabile, che la directory non sia stata aggiornata con il valore della lunghezza reale:

```
FUNCTION FileLength& (handle%)
      '______________________________
      '  Restituisce la lunghezza di un file
      '______________________________
      DIM reg AS RegTypeX, hiSeek%, loSeek%
      reg.ax = &H4201            ' usa il metodo AL = 1
      reg.bx = handle%           ' CX=0, DX=0
      InterruptX &H21, reg, reg
      hiSeek% = reg.cx           ' ricorda il valore corrente
      loSeek% = reg.dx           ' del puntatore al file

      reg.ax = &H4202            ' usa il metodo AL = 2
      reg.cx = 0                 ' sposta il puntatore alla fine del file
      reg.dx = 0
      InterruptX &H21, reg, reg
      ' il valore corrente del puntatore è la lunghezza del file
      FileLength& = CVL(MKI$(reg.dx) + MKI$(reg.cx))

      reg.ax = &H4200            ' usa il metodo AL = 0
      reg.cx = hiSeek%           ' ripristina il puntatore originario
      reg.dx = loSeek%
      InterruptX &H21, reg, reg
END FUNCTION
```

IL FLUSHING DEI BUFFER

Giacché siamo in argomento, è bene ricordare che le operazioni di scrittura su file da parte di un programma comportano il trasferimento dei dati in un buffer, da cui solo in un secondo momento saranno portate su disco. In altre parole, si tratta di una scrittura *logica*, che non deve essere confusa con la scrittura *fisica* sul mezzo magnetico. Normalmente il Dos esegue la scrittura fisica solo quando il buffer (512 byte) è pieno, oppure quando il file è chiuso, o ancora quando non vi sono buffer disponibili e il programma reclama un buffer libero:

in tutti gli altri casi, possono passare anche alcuni secondi (o addirittura minuti) dalla scrittura logica alla corrispondente operazione fisica, e se il sistema subisce un crash oppure si verifica una assenza di alimentazione, è facile immaginarne le conseguenze: le ultime informazioni scritte dal programma non saranno state trasferite su disco, né la directory sarà stata aggiornata con i nuovi valori di lunghezza e data di aggiornamento del file.

A partire dalla versione 3.30, il Dos include un servizio dell'interrupt 21 che appunto eseguigue il flushing di un file; si tratta della funzione 68h *Commit File*, che richiede in BX l'handle del file. Il listato seguente riporta una procedura che appunto esegue tale operazione:

```
SUB FileFlush (handle%)
	'___________________________
	' Esegue il flushing di un file aperto
	' IMPORTANTE: richiede Dos 3.30 o superiore
	'___________________________
	DIM reg AS RegTypeX
	reg.ax = &H6800
	reg.bx = handle%
	InterruptX &H21, reg, reg
END SUB
```

Se siamo certi che il nostro programma è eseguito sotto Dos 3.30 o una versione successiva va quindi tutto bene, ma cosa fare se vogliamo rendere il nostro programma compatibile anche con le versioni precedenti ? Se vogliamo eseguire il flushing con qualunque versione del Dos, possiamo sfruttare un servizio poco noto ed usato, ovvero la funzione 45h *Duplicate Handle*: questa funzione crea un alias (un duplicato, appunto) per un handle di file, e tutte le operazioni di lettura, scrittura e spostamento puntatore eseguite con questo nuovo handle avranno effetto anche sull'handle originario, e viceversa. Il punto più interessante per la nostra discussione è che se chiudiamo il file relativo all'handle duplicato il file originario resta aperto ma i buffer vengono svuotati e la directory aggiornata con la lunghezza attuale del file, il che è esattamente quello che volevamo. Il listato seguente riporta la procedura **FileFlush2**, che funziona con tutte le versioni Dos a partire dalla 2.0

```
SUB FileFlush2 (handle%)
	'___________________________
	' Esegue il flushing di un file aperto
	' Funziona con tutte le versioni Dos
	'___________________________
	DIM reg AS RegTypeX
	reg.ax = &H4500                 ' crea un handle duplicato
	reg.bx = handle%
	InterruptX &H21, reg, reg
	IF (reg.flags AND 1) = 0 THEN   ' se non vi sono errori
		reg.bx = reg.ax         ' AX contiene il nuovo handle
		reg.ax = &H3E00         ' chiudi il nuovo file
		InterruptX &H21, reg, reg
	END IF
END SUB
```

Nel decidere quale delle due routine utilizzare nei propri programmi si tenga a mente che **FileFlush2** presenta alcuni inconvenienti di cui è bene essere al corrente: (1) potrebbe fallire a causa di indisponibilità di handle, e (2) il programma non rischia di perdere l controllo sul file in un ambiente multitasking o di rete locale. In definitiva, la soluzione migliore è di testare la versione del sistema operativo ed usare **FileFlush2** solo se risulta essere precedente la 3.30.

AUMENTARE IL NUMERO DI HANDLE

C'è un aspetto del sistema operativo sul quale ho riscontrato una grande confusione, anche tra programmatori non alle prime armi. Come tutti sanno, la direttiva FILES in CONFIG.SYS serve ad impostare il massimo numero di file che possono essere contemporaneamente aperti nel sistema. Non si tratta quindi del numero di file che è possibile aprire da un singolo programma, ma di un valore che tiene conto di tutti i programmi attivi. In un sistema formalmente mono-task come il Dos questa distinzione non dovrebbe aver senso, ma in pratica occorre tener conto dei programmi TSR, per non parlare di Windows o anche più semplicemente della possibilità che un programma ne lanci un altro attraverso una istruzione di SHELL.

In realtà accade che, indipendentemente dal valore di FILES, esiste *comunque* un limite al numero di file che un programma può aprire, e per la precisione si tratta di 20 file. In questo numero sono già compresi i 5 handle che per default il sistema operativo apre per il programma (lo standard input, lo standard output, lo standard error, la stampante e la porta seriale). Cosa succede accade se il nostro programma deve trattare con più di 15 file su disco?

La prima, ovvia, soluzione è quella di chiudere gli handle di default che non ci interessano: ad esempio, potremmo chiudere lo standard error (handle = 2) quasi mai usato, il canale per la stampante parallela (handle = 4) se il nostro programma non ne richiede l'uso, e lo standard auxiliary output (handle = 3), anch'esso raramente usato. In questo modo possiamo recuperare fino a tre handle, e se essi sono sufficienti al nostro programma abbiamo risolto il problema.

Come fare se invece la nostra applicazione richiede venti, trenta o più handle ? A partire dal Dos 3.3 esiste un servizio che permette di superare il limite dei 20 handle per applicazione; si tratta della funzione 67h, che richiede in BX il numero degli handle desiderati. Il Dos provvede a creare una tavola in memoria delle dimensioni adeguate e ad inizializzarla per contenere gli handle relativi al programma che ha invocato la funzione stessa. Perché questo

servizio funzioni correttamente è quindi necessario che vi sia memoria sufficiente per creare la tavola, e perciò il BASIC deve provvedere a rilasciare parte della memoria che il sistema operativo gli ha assegnato all'inizio dell'esecuzione del programma. La seguente procedura provvede a tutto questo:

```
SUB SetMaxHandle(numHandles%)
    '________________________________
    ' Imposta un nuovo limite al numero di file
    ' che possono essere aperti dal programma
    '________________________________
    DIM reg AS RegTypeX, temp&
    ' libera la memoria per la nuova tavola degli handle
    temp& = SETMEM(-numHandles% * 2 - 32)
    reg.ax = &H6800
    reg.bx = numHandles%
    InterruptX &H21, reg, reg
END SUB
```

LEGGERE LA DATA E L'ORARIO DI UN FILE

Il BASIC standard non dispone di alcun sistema per ricavare la data di un file su disco; in molti casi questa informazione è importantissima, ad esempio per controllare l'ultimo aggiornamento di un archivio oppure per accertarsi di adoperare l'ultima versione di un programma. A tale scopo il Dos mette a disposizione il servizio 57h dell'interrupt 21; per determinare la data di un file esistente occorre memorizzare il valore 0 in AL e l'handle del file in BX. Se non vi sono errori, cioé se il valore dell'handle è corretto, la data del file è restituita in DX ed il suo orario in CX. Il processo di estrazione dei risultati non è però completamente indolore, in quanto sia la data che l'ora sono compattati in una word di 16 bit con un formato abbastanza particolare:

```
DX = data
        i bit 0-4 contengono il giorno (1-31)
        i bit 5-8 contengono il mese (1-12)
        i bit 9-15 contengono l'anno (relativo al 1980)
CX = orario
        i bit 0-4 contengono il valore dei secondi diviso 2 (quindi nell'intervallo 0-29)
        i bit 5-10 contengono il valore dei minuti (0-59)
        i bit 11-15 contengono il valore delle ore (0-23)
```

Gli anni sono relativi al 1980 (primo anno dell'era del personal computer Ms-Dos), per cui ad esempio il 1994 sarà memorizzato come "14"; inoltre per motivi di spazio gli orari sono approssimati ai 2 secondi, per cui due file creati a meno di due secondi uno dall'altro possono essere registrati con lo stesso orario. Poiché il BASIC non dispone di istruzioni per la manipolazione diretta dei bit è necessario ricorrere ad alcune tecniche non sempre particolarmente elegan-

ti. Ad esempio, per isolare il valore dei secondi, occorre porre a zero il valore dei bit 5-15, cosa che in Assembly si ottiene con l'istruzione:

```
AND     CX,0000000000011111B
```

purtroppo il BASIC non dispone delle costanti binarie, e siamo costretti a scrivere qualcosa del genere:

```
secondi% = (reg.cx AND &H1F) * 2
```

Le cose sono sono leggermente più complicate nel caso dei minuti, i cui bit corrispondenti sono nel mezzo della word; in questo caso occorre shiftare la word di cinque bit verso destra prima di usare l'operatore AND per estrarre il valore vero e proprio. Apparentemente niente di complicato: basta dividere per 32 il valore del registro CX, e scrivere qualcosa del genere

```
minuti% = (reg.cx \ 32) AND &HF
```

Purtroppo questa tecnica non funziona, o meglio, non funziona correttamente in tutte le occasioni; vediamo perché. Quando il BASIC esegue la divisione intera, arrotonda il risultato verso l'intero più piccolo, per cui ad esempio risulta che &H4001 \ 2 = &H2000; se però il bit più significativo del valore a 16 bit è diverso da zero, il BASIC interpreta il numero come un valore negativo, e in tal caso il troncamento verso l'intero immediatamente più piccolo non sortisce il risultato atteso. Ad esempio, (&H8001 \ 2) è uguale a &HC001, e non a &4000 come si è portati ad immaginare; del resto, se abbandoniamo la notazione esadecimale e passiamo ai valori equivalenti decimali, è evidente che se dividiamo un numero negativo (&H8001 = -32767) per 2 che è un numero positivo, dobbiamo forzatamente ottenere un risultato negativo (&HC001 = -16383).

Torniamo al nostro problema: se vogliamo isolare il valore dei minuti dobbiamo *prima* mascherare tutti gli altri bit a zero, e *dopo* shiftare il numero verso destra mediante divisione intera; in tal modo, l'operazione di AND raggiunge il duplice scopo di rendere il numero positivo e troncare eventuali parti frazionarie

```
minuti% = (reg.cx AND &H7E0) \ 32)
```

Per estrarre il numero delle ore si pone un ulteriore problema, in quanto non possiamo rendere il numero positivo prima della divisione, visto che il bit più significativo (il bit 15) appartiene al valore da estrarre e non può essere azzerato come nei caso dei minuti. Occorrono allora due operazioni di AND, una prima della divisione per eliminare la possibilità di risultati frazionari (che come abbiamo detto in precedenza potrebbero essere troncati in modo a noi sconveniente) ed una dopo la divisione per azzerare tutti i bit a sinistra del valore prodotti dalla divisione di un numero negativo. Il risultato, poco leggibile ma corretto, è il seguente:

```
hour% = ((reg.cx AND &HF800) \ &H800 ) AND &H1F)
```

Analoghe considerazioni si possono fare riguardo allo scompattamento della data in DX. Ecco il listato della subroutine **FileDate**, che restituisce appunto data e ora di un file interpretando i risultati restituiti dall'interrupt del Dos:

```
SUB FileDate (handle%, year%, month%, day%, hour%, minute%, second%)
      '____________________________
      ' Determina la data e l'ora di creazione di un file
      ' restituisce i valori attraverso gli argomenti
      '   in caso di errore YEAR è impostato uguale a zero
      '____________________________
      DIM reg AS RegTypeX
      reg.ax = &H5700
      reg.bx = handle%
      InterruptX &H21, reg, reg
      IF reg.flags AND 1 THEN   ' se il flag di carry è impostato ad uno
              year% = 0                      ' segnala errore
      ELSE
              ' isola le tre componenti dell'orario
              second% = (reg.cx AND &H1F) * 2
              minute% = (reg.cx AND &H7E0) \ &H20
              hour% = ((reg.cx AND &HF800) \ &H800 ) AND &H1F
              ' isola le tre componenti della data
              day% = reg.dx AND &H1F
              month% = (reg.dx AND &H1E0) \ &H20
              year% = ((reg.dx AND &HFE00) \ &H200 ) AND &H7F
      END IF
END SUB
```

MODIFICARE LA DATA E L'ORARIO DI UN FILE

Ovviamente il Dos offre anche la possibilità di modificare il valore della data e ora di un file; tra i programmi che sfruttano questa possibilità vi sono tutte le utility di *touch*, solitamente fornite con i sistemi di MAKE; tra i programmi meno "leciti" vi sono alcuni virus che dopo aver infettato un file usando i servizi standard del Dos ripristinano la sua data di creazione originale, in modo da non dare all'utente una traccia troppo evidente della loro presenza.

Tempo fa ho scritto una utility di TOUCH che modifica appunto la data e l'orario di creazione di un file; ecco uno degli usi meno ortodossi: i clienti della SoftWhale sono sparsi in tutta Italia e la maggior parte dell'assistenza tecnica è richiesta per via telefonica. Per capire subito quale versione del software è usata dal cliente, assegno la data di spedizione ai file, e faccio inoltre in modo che l'orario rispecchii il numero di versione (ad es. 1.04 oppure 2.10). Allora è sufficiente chiedere al cliente di eseguire un comando DIR per capire quali problemi posso aspettarmi e come possono essere risolti, con evidente risparmio di tempo per entrambi.

Per questa funzione il Dos mette a disposizione il sottoservizio 1 dello stesso servizio 57h appena visto; in questo caso il registro AL deve contenere il valore 1 (per indicare che intendiamo scrivere la data e non leggerla), il registro BX

deve contenere l'handle del file, il registro DX la data da assegnare al file, il registro CX l'orario: sia la data che l'orario devono essere "impacchettati" in una sola word di 16 bit, per cui dobbiamo risolvere problemi analoghi a quelli visti in precedenza.

Poiché il valore dei secondi occupa nel registro CX i bit meno significativi, possiamo a comporre il valore di tale registro in questo modo:

```
reg.cx = (second% \ 2) AND &H1F
```

Passiamo al valore dei minuti: quando si impacchettano dei campi codificati a bit l'operatore da usare è l'OR booleano, *dopo* però aver eseguito uno shift verso sinistra del numero di bit necessari; poiché il BASIC non supporta gli shift dobbiamo ricorrere alla moltiplicazione;

```
reg.cx = reg.cx OR (minute% AND &H3F) * &H20
```

Arrivati però al valore delle ore, le cose si complicano; infatti, il valore degli anni deve essere shiftato a sinistra di 11 bit, ossia moltiplicato per &H800 (=2048), per cui saremmo tentati di scrivere

```
reg.cx = reg.cx OR (hour% AND &H1F) * &H800
```

che però provoca un errore di overflow aritmetico quando il numero delle ore è maggiore di 15; in tal caso infatti la moltiplicazione fornisce un risultato che non può essere espresso come integer a 16 bit. Occorre allora forzare la moltiplicazione a 32 bit, usando la funzione CLNG oppure una costante di tipo LONG (notate il simbolo "&" finale nel secondo rigo):

```
reg.cx = reg.cx OR CLNG(hour% AND &H1F) * &H800
reg.cx = reg.cx OR hour% AND &H1F) * &H800&
```

ma l'overflow rimane comunque, causato dal tentativo di assegnare il risultato dell'operazione OR (un valore a 32 bit) ad una variabile integer. Per evitare il problema è necessario riconvertire a 16 bit il risultato della moltiplicazione, in questo modo

```
reg.cx = reg.cx OR CVI(MKL$((hour% AND &H1F) * &H800&))
```

Il seguente listato riassume tutti i contorcimenti programmatori necessari per arrivare al risultato desiderato:

```
SUB SetFileDate (handle%, year%, month%, day%, hour%, minute%, second%)
      '______________________
      ' Imposta la data e l'ora di creazione di un file
      '     in caso di errore pone YEAR = 0
      '______________________
      DIM reg AS RegTypeX
      ' componi le tre componenti dell'orario
      reg.cx = (second% \ 2) AND &H1F
      reg.cx = reg.cx OR (minute% AND &H3F) * &H20
      reg.cx = reg.cx OR CVI(MKL$((hour% AND &H1F) * &H800&))
      ' componi le tre componenti della data
      reg.dx = day% AND &H1F
      reg.dx = reg.dx OR (month% AND &HF) * &H20
      reg.dx = reg.dx OR CVI(MKL$(((year% - 1980) AND &H7F)* &H200&))
      ' chiama il Dos
      reg.ax = &H5701          ' AL = 1 per impostare la data
      reg.bx = handle%
```

```
        InterruptX &H21, reg, reg
        ' azzera YEAR in caso di errore
        IF reg.flags AND 1 THEN year% = 0 ' segnala errore azzerando YEAR
    END SUB
```

Ovviamente, se un programma usa questo servizio per impostare la data di un file aperto, il Dos eviterà di influenzare la data e l'ora quando il file stesso sarà chiuso (in caso contrario sarebbe tutto inutile).

GLI ATTRIBUTI DI UN FILE

Come per la data, è anche possibile leggere e impostare gli attributi di un file. In questo caso il servizio Dos interessato è il 43h, che come il 57h dispone di due sottoservizi: se AL=0 il Dos leggerà gli attributi del file, mentre se AL=1 il Dos imposterà gli attributi con il valore desiderato. In entrambi i casi il registro CX conterrà gli attributi del file (restituiti dal Dos se AL=0 oppure passati al Dos dal programma se AL=1), ed infine DS:DX deve puntare ad una stringa ASCIIZ che contiene il nome del file, eventualmente completo del percorso.

Gli attributi in CX sono codificati come segue:

bit 0 a sola lettura

bit 1 file nascosto

bit 2 file di sistema

bit 3 etichetta di volume

bit 4 sottodirectory

bit 5 file di archivio

per la maggior parte dei file su disco il valore di CX sarà 32, corrispondente al solo attributo di archivio impostato; questo è il bit che viene attivato automaticamente dal Dos ad ogni modifica del file, e che viene disattivato dai comandi BACKUP oppure XCOPY /M.

Il bit 3 individua il file che funge da etichetta di volume: il Dos infatti memorizza l'etichetta (creata con LABEL o FORMAT) nella directory radice come se fosse un file lungo zero byte e l'unica differenza rispetto ad un file normale è in questo bit di attributo: questo spiega perché, ad esempio, le etichette di volume sono lunghe massimo undici caratteri: si tratta della lunghezza del campo che il Dos mette a disposizione per la combinazione nomefile+estensione (8+3). Non si tenti di modificare direttamente lo stato di questo bit. (NOTA: a partire da MsDos 4.0 l'etichetta è conservata nel boot record del drive a cui si rifierisce).

Il bit 4 è particolare, in quanto individua una sottodirectory; a differenza della directory radice, infatti, le sottodirectory altro non sono che particolari file, composti da tanti record di 32 byte quanti sono i file contenuti nella sottodirectory stessa, e dal punto di vista "fisico" l'unica differenza tra un file e una sottodirectory è fornita proprio da questo bit. Ovviamente il sistema operativo si rifiuterà di convertire una directory in un file o un file in una directory: in entrambi i casi le conseguenze sarebbero disastrose e comporterebbero perdite di dati. Nulla però impedisce di impostare il bit di "file nascosto" anche per le sottodirectory, la qual cosa fornisce un buon sistema di protezione dagli occhi indiscreti.

Le routine seguenti mettono in pratica quanto detto, e permettono sia di leggere che di modificare gli attributi di un file esistente.

```
FUNCTION FileAttribute% (filename$)
      '________________________________
      ' Restituisce gli attributi di un file
      '   oppure -1 in caso di errore (es. file inesistente)
      '________________________________
      DIM reg AS RegTypeX
      DIM buffer AS STRING * 80
      buffer = filename$ + CHR$(0)          ' crea una stringa ASCIIZ
      reg.ax = &H4300                       ' AL=0 per leggere gli attributi
      reg.ds = VARSEG(buffer)               ' indirizzo del nome in DS:DX
      reg.dx = VARPTR(buffer)
      InterruptX &H21, reg, reg
      IF reg.flags AND 1 THEN
              FileAttribute% = -1           ' restituisce -1 in caso di errore
      ELSE
              FileAttribute% = reg.cx       ' altrimenti il valore di CX
      END IF
END FUNCTION
SUB SetFileAttribute (filename$, attributes%)
      '________________________________
      '       Imposta gli attributi di un file
      '       in caso di errore, restituisce ATTRIBUTES% = -1
      '________________________________
      DIM reg AS RegTypeX
      DIM buffer AS STRING * 80
      buffer = filename$ + CHR$(0)          ' crea una stringa ASCIIZ
      reg.ax = &H4301                       ' AL=1 per modificare gli attributi
      reg.cx = attributes%
      reg.ds = VARSEG(buffer)               ' indirizzo del nome in DS:DX
      reg.dx = VARPTR(buffer)
      InterruptX &H21, reg, reg
      IF reg.flags AND 1 THEN attributes% = -1  ' in caso di errore
END SUB
```

IL BIOS

Anche se il sistema operativo fornisce numerosi servizi che possono essere sfruttati dai programmi applicativi, per alcune funzioni è necessario scavalcare il Dos e interfacciarsi direttamente con il BIOS; può trattarsi ad esempio del controllo della scheda video, della lettura di alcuni parametri di configurazione oppure dell'accesso ai dischi a basso livello. In questo capitolo passeremo in rassegna tutti i principali servizi offerti dal BIOS che possono risultare utili nei programmi BASIC.

IL BIOS VIDEO

L'interrupt 10 esadecimale è riservato dal BIOS ai servizi del video; le routine che servono questo interrupt sono situate in parte nel ROM BIOS che si trova nella parte alta del primo magabyte di indirizzamento dei processori 80x86 e in parte nel BIOS che viene installato con le schede EGA e VGA. Molti dei servizi video non sono di grande utilità per i programmatori BASIC, in quanto si può accedere ad essi mediante le istruzioni standard del linguaggio, pertanto mi limiterò ad illustrare tutto quello che non può essere ottenuto in BASIC "puro".

Due servizi dell'interrupt 10h sono molto utili poiché permettono di far scorrere verso l'alto o verso il basso il contenuto di una porzione rettangolare di schermo; nel passare i valori al BIOS occorre però tenere presente che quest'ultimo considera il primo carattere in alto a sinistra come (0,0) e non

(1,1) come accade in BASIC. Ecco allora la procedura che fa scorrere una porzione di testo verso l'alto:

```
SUB ScrollUp (top%, left%, bottom%, right%, lines%, colorCode%)
      '______________________________
      ' Scorrimento di una finestra verso l'alto
      '______________________________
      DIM reg AS RegType
      ' AH deve contenere il numero del servizio, AL il numero di righe
      reg.ax = &H600 + (lines% AND 255)
      ' BH deve contenere l'attributo di colore
      reg.bx = (colorCode% AND 255) * 256
      ' CH-CL devono contenere la riga/colonna dell'angolo superiore sinistro
      reg.cx = (top% - 1) * 256 + (left% - 1)
      ' DH-DL devono contenere la riga/colonna dell'angolo inferiore destro
      reg.dx = (bottom% - 1) * 256 + (right% - 1)
      Interrupt &H10, reg, reg
END SUB
```

La procedura **ScrollDown** è perfettamente simile, con la sola differenza che AH deve contenere il valore 7 anziché 6; sfruttando questa somiglianza possiamo anche usare un'unica procedura, con la convenzione che se **lines%** è positivo lo scorrimento è verso l'alto, mentre se è negativo lo scorrimento è verso il basso:

```
SUB ScrollWindow (top%, left%, bottom%, right%, lines%, colorCode%)
      '______________________________
      ' Scorrimento di una finestra verso l'alto(se LINES% > 0)
      '   o verso il basso (se LINES% < 0)
      '______________________________
      DIM reg AS RegType
      ' AH deve contenere il numero del servizio, AL il numero di righe
      IF lines% > 0 THEN
              reg.ax = &H600 + (lines% AND 255)
      ELSE
              reg.ax = &H700 + (-lines% AND 255)
      END IF
      ' BH deve contenere l'attributo di colore
      reg.bx = (colorCode% AND 255) * 256
      ' CH-CL devono contenere la riga/colonna dell'angolo superiore sinistro
      reg.cx = (top% - 1) * 256 + (left% - 1)
      ' DH-DL devono contenere la riga/colonna dell'angolo inferiore destro
      reg.dx = (bottom% - 1) * 256 + (right% - 1)
      Interrupt &H10, reg, reg
END SUB
```

Una particolarità dei servizi 6 e 7 è di poter cancellare il contenuto dell'area indicata quando si passa un valore nullo in AL; pertanto la procedura ScrollWindow può anche essere usata per cancellare il contenuto di una finestra, usando caratteri del colore indicato

```
      ScrollWindow 3, 5, 20, 60, 0, &H70
```

in **ScrollWindow** il codice del colore è un byte i cui quattro bit meno significativi indicano il colore per il testo, e i quattro bit più significati il colore dello sfondo. Ad esempio, per cancellare il contenuto della finestra ed

inizializzare i colori a rosso brillante (codice 12) su sfondo blu (codice 1) si deve eseguire

```
ScrollWindow 3, 5, 20, 60, 0, &H1C
```

Il servizio 0F restituisce il modo video corrente, e può essere utile per scrivere delle subroutine che funzionano indipendentemente dal modo video testo o grafico

```
SUB GetVideoMode (videoMode%, columns%, page%)
        '______________________________
        ' Restituisce il modo video corrente, il numero di
        ' colonne visualizzate e la pagina video corrente
        '______________________________
        DIM reg AS RegType
        reg.ax = &HF00
        Interrupt &H10, reg, reg          ' restituisce:
        videoMode% = (reg.ax AND 255)     ' modo video in AL
        columns% = (reg.ax \ 256)         ' colonne in AH
        page% = (reg.bx \ 256)            ' numero pagina in BH
END SUB
```

Un piccolo problema è dato dal fatto che il valore restituito in **videoMode%** non corrisponde direttamente al codice con cui il BASIC individua i modi grafici, ad es. nel comando SCREEN. Ecco una tavola di conversione (notare che alcuni modi grafici non hanno un corrispondente in BASIC):

BIOS	BASIC
01H	SCREEN 0, WIDTH 40
03H	SCREEN 0 (schede colore)
04H	SCREEN 1
06H	SCREEN 2
07H	SCREEN 0 (schede monocromatiche)
0DH	SCREEN 7
0EH	SCREEN 8
0FH	SCREEN 10
10H	SCREEN 9
11H	SCREEN 11
12H	SCREEN 12
13H	SCREEN 13

Per le schede EGA e VGA è possibile impostare il colore del bordo, usando il servizio 10h sottofunzione 1:

```
SUB SetBorderColor (colorCode%)
        '______________________________
        ' Imposta il colore del bordo (solo EGA e VGA)
        '______________________________
        DIM reg AS RegType
        reg.ax = &H1001             ' AH = 10h, AL = 1
        reg.bx = colorCode% * 256 ' BH = codice del colore
        Interrupt &H10, reg, reg
END SUB
```

mentre la sottofunzione 8 restituisce il colore corrente del bordo

```
FUNCTION GetBorderColor%
        '______________________________
```

```
        ' Restituisce il colore del bordo (solo EGA e VGA)
        '____________________________
        DIM reg AS RegType
        reg.ax = &H1008            ' AH = 10h, AL = 8
        Interrupt &H10, reg, reg
        ' restituisce il colore del bordo in BH
        GetBorderColor% = (reg.bx \ 256) AND 255
END FUNCTION
```

PARAMETRI DI CONFIGURAZIONE

Gli interrupt 11h e 12h sono particolari, in quanto entrambi forniscono un'unico servizio e pertanto non è necessario specificare un valore in AH. L'interrupt 11h restituisce in AX delle informazioni sulla configurazione del sistema; tali informazioni sono codificate in bit, come segue

```
bit     significato
0       = 1 se il computer dispone di uno o più unità floppy
1       = 1 se è installato un coprocessore matematico
2-3     indica la quantità di RAM sulla cheda madre:
                00      16K             10      48K
                01      32K             11      64K
        sui PS/2 il bit 2 è impostato ad 1 se il sistema dispone di un mouse
4-5     modo video iniziale
                01      modo testo a colori 40x25
                10      modo testo a colori 80x25
                11      modo testo monocromatico 80x25
6-7     numero di unità floppy installate (se il bit 0 è non zero)
                00      uno             10      tre
                01      due             11      quattro
9-11    numero di porte seriali RS-232 installate
12      = 1 se il computer dispone di una porta giochi (per joystick)
14-15   numero di porte parallele installate
```

Alcune di queste informazioni non sono di grande utilità per i programmi applicativi e possono essere tralasciate (ad es. la quantità di memoria su scheda madre oppure il modo video iniziale); per le altre possiamo creare una procedura BASIC

```
SUB GetConfigurationInfo (floppy%, mathCo%, serial%, joystick%, parallel%)
        '____________________________
        ' Restituisce le seguenti informazioni sulla configurazione
        '   FLOPPY%     il numero di unità floppy installate
        '   MATHCO%     -1 se è installato un coprocessore matematico
        '   SERIAL%     il numero di porte seriali RS-232
        '   JOYSTICK%   -1 se un joystick è collegato al sistema
        '   PARALLEL%   il numero di porte parallele
        '____________________________
        DIM reg AS RegType
        Interrupt &H11, reg, reg
        ' se il bit 0 è diverso da zero, i bit 6-7 contengono il numero
        ' di unità floppy installate
        IF reg.ax AND 1 THEN
                floppy% = (reg.ax AND &HC0) \ 64 + 1
        ELSE
                floppy% = 0
        END IF
        ' il bit 1 indica la presenza del coprocessore matematico
        mathCo% = (reg.ax AND 2) <> 0
```

```
        ' il numero di porte seriale è conservato nei bit 9-11
        serial% = (reg.ax AND &HE00) \ 512
        ' il bit 12 indica la presenza di un joystick
        joystick% = (reg.ax AND &H1000) <> 0
        ' il numero di porte parallele è conservato nei bit 14-15
        ' poiché il bit 15 è il bit che indica il segno del numero
        ' in AX, è necessario forzare le operazioni sui LONG
        parallel% = (reg.ax AND &HC000&) \ 16384
END SUB
```

Se il nostro programma non richiede tutte queste informazioni, possiamo anche costruire delle funzioni, ad esempio

```
FUNCTION SerialPorts%
        '___________________________
        ' Restituisce il numero di porte seriali
        '___________________________
        DIM reg AS RegType
        Interrupt &H11, reg, reg
        SerialPorts% = (reg.ax AND &HE00) \ 512
END FUNCTION
```

L'unica funzione dell'interrupt 12h è di restituire in AX la quantità di memoria convenzionale in kilobyte trovata al bootstrap; questo valore non tiene conto della eventuale memoria estesa o superiore, e non è mai superiore al valore 640

```
FUNCTION RamMemory%
        '___________________________
        ' Restituisce la quantità di memoria convenzionale
        '___________________________
        DIM reg AS RegType
        Interrupt &H12, reg, reg
        RamMemory% = reg.ax
END FUNCTION
```

Inoltre, il valore restituito non tiene conto della eventuale Extended BIOS Area, una zona di memoria allocata al bootstrap dal BIOS dei PS/2 e di alcuni modelli di BIOS per 80286 o superiori per conservare alcune variabili; per questo motivo in alcuni casi la funzione RamMemory% può restituire un valore inferiore a 640.

I SERVIZI BIOS PER I DISCHI

I programmi scritti in un linguaggio ad alto livello interagiscono con le informazioni memorizzate su disco usando i nomi di file e directory e lasciando al sistema operativo il compito di tradurre le nostre direttive in termini di cilindri, tracce e settori. Non vi sono quindi molti motivi validi per accedere direttamente ai dischi tramite l'interrupt 13h del BIOS e scavalcando il sistema operativo, ma vi sono alcuni servizi accessori che possono rivelarsi utili anche nei programmi applicativi.

Una delle caratteristiche più interessanti del BIOS è di comunicare gli errori attraverso il flag di carry e un codice di errore restituito in AH, senza provocare un errore critico se il drive non è pronto o il dischetto non è formattato. Possiamo allora usare il servizio 2 dell'interrupt 13h per leggere un qualsiasi settore di un floppy, e verificare che l'operazione non restituisca un errore

```
FUNCTION IsDriveReady% (drive$)
        '______________________________
        ' Restituisce -1 se un driver per floppy contiene un
        ' dischetto formattato e pronto per la lettura
        '  drive$ deve essere "A" oppure "B"
        '______________________________
        DIM reg AS RegTypeX, count%
        DIM buffer%(255)                ' il buffer per la lettura (512 byte)
        ' controlla che il parametro indichi un drive per floppy
        IF INSTR("AaBb", drive$) = 0 OR drive$ = "" THEN EXIT FUNCTION
        ' per accertarsi che l'errore non sia dovuto al fatto che il
        ' motore del drive era spento, occorre ripetere le seguenti
        ' operazioni per almeno tre volte
        FOR count% = 1 TO 3
                reg.ax = &H201                  ' AH = servizio 2, AL = numero settori
                reg.cx = 1                      ' CH = cilindro, CL = settore
                ' DL deve contenere il numero del drive (A=0, B=1), e DH il
                ' numero della testina (possiamo usare zero)
                reg.dx = ASC(UCASE$(drive$)) - 65
                reg.es = VARSEG(buffer%(0))     ' ES:BX = l'indirizzo del buffer
                reg.bx = VARPTR(buffer%(0))
                InterruptX &H13, reg, reg
                ' esci se non vi è stato errore
                IF (reg.flags AND 1) = 0 THEN EXIT FOR
        NEXT
        IsDriveReady% = (reg.flags AND 1) = 0
END FUNCTION
```

Per comprendere il funzionamento della routine IsDriveReady% occorre sapere che il BIOS vede i dischetti come un insieme di settori, ciascuno lungo 512 byte, a cui si può accedere indicandone la posizione in termine di cilindro e testina di lettura (nel caso dei floppy disk vi sono due sole testine, numerate 0 e 1). Nel nostro caso possiamo leggere un settore qualsiasi del drive, ma l'operazione deve essere ripetuta almeno tre volte (vedi il ciclo FOR con indice count%) perché i primi due tentativi potrebbero fallire se il motore del drive non è in funzione al momento della prima richiesta.

I drive per floppy installati sui computer AT, 386 e 486 (ma anche sui sistemi XT più recenti) permettono di sapere se un dischetto è stato cambiato dall'ultima operazione di lettura o scrittura; in tal modo un programma applicativo può determinare se è necessario effettuare nuovamente i controlli sul disco (ad esempio per verificare l'etichetta di volume o la presenza di qualche file particolare):

```
FUNCTION DiskChanged% (drive$)
        '______________________________
        ' Restituisce -1 se il dischetto nel drive indicato è
        ' stato sostituito
        '______________________________
```

```
        DIM reg AS RegType
        ' controlla che il parametro indichi un drive per floppy
        IF INSTR("AaBb", drive$) = 0 OR drive$ = "" THEN EXIT FUNCTION
        reg.ax = &H1600                      ' servizio 16h in AH
        reg.dx = ASC(UCASE$(drive$)) - 65 ' numero drive in DL
        Interrupt &H13, reg, reg
        ' il disco è stato sostituito se il flag di carry è impostato
        ' e il registro AH contiene il valore 6
        DiskChanged% = (reg.flags AND 1) <> 0 AND (reg.ax AND &HFF00) = &H600
END FUNCTION
```

Si noti che se la funzione DiskChanged% restituisce -1 non significa che sicuramente il dischetto è stato scambiato, ma soltanto che è stato aperto e richiuso lo sportellino del drive; inoltre si tenga presente che in alcuni lettori questa funzione non è attiva.

LE VARIABILI DEL BIOS

Il BIOS conserva numerose informazioni sul sistema in un'area della memoria convenzionale e i programmi applicativi possono trovare conveniente, in alcuni casi, scavalcare persino lo stesso BIOS e accedere direttamente a quest'area, per leggere e persino modificare questi valori. Poiché alcune informazioni sono organizzate in word, può essere utile definire una funzione per accedere direttamente al valore

```
FUNCTION PeekW% (offset%)
        PeekW% = CVI(CHR$(PEEK(offset%)) + CHR$(PEEK(offset% + 1)))
END FUNCTION
```

La BIOS Area comincia all'indirizzo 0000:0400: ecco una lista delle locazioni più interessanti ed utili.

```
400 (4 word) Indirizzi delle porte seriali
```

Ciascuna di queste word contengono l'indirizzo delle porte RS-232 eventualmente installate nel sistema; quindi PeekW%(&H400) restituisce l'indirizzo della porta COM1, PeekW%(&H404) l'indirizzo di COM3 e così via. Poiché il BASIC è in grado di accedere soltanto alle prime due porte seriali, manipolando questi indirizzi è possibile indirizzare anche COM3 e COM4. Ecco un programma che apre la porta seriale COM3

```
DEF SEG = 0
' conserva l'indirizzo della porta COM1
byte1% = PEEK(&H400): byte2% = PEEK(&H401)
' memorizza in 0400-0401 l'indirizzo della porta COM3
POKE &H400, PEEK(&H404): POKE &H401, PEEK(&H405)
DEF SEG
' apri COM3 come se fosse COM1
handle# = FREEFILE
OPEN "COM1:1200" FOR OUTPUT AS #handle
...
' al termine della operazioni ripristina l'indirizzo di COM1
DEF SEG = 0
```

```
    POKE &H400, byte1%: POKE &H401, byte2%
    DEF SEG
408 (4 word) Indirizzi delle porte parallele
```

Queste quattro word conservano l'indirizzo delle quattro porte parallele LPT1-LPT4; usando il medesimo sistema appena spiegato è possibile accedere da BASIC ad una qualunque delle stampanti parallele collegate al computer.

```
410 (word) Configurazione del sistema
```

Questo è il valore restituito anche dall'interrupt 11h, come abbiamo visto in questo stesso capitolo; i programmi applicativi possono quindi accedere alle stesse informazioni leggendo questi due byte, ma è sconsigliabile modificarli.

```
413 (word) Dimensione della memoria convenzionale
```

Questa è la stessa informazione restituita dall'interrupt 12h; anche in questo caso un programma potrebbe ottenere il valore con una lettura diretta di queste locazioni, ma il suo contenuto non dovrebbe essere modificato.

```
417 (word) Stato dei tasti di shift
```

Questi due byte riportano lo stato corrente dei tasti di shift (cioè ALT, CTRL e i due SHIFT) e dei tasti di lock (CapsLock, NumLock, ScrollLock e Insert); il primo byte è restituito da una chiamata al servizio 02 dell'interrupt 16h, ma può anche essere direttamente alterato dai programmi per modificare lo stato di lock, usando una tecnica descritta nel capitolo dedicato alla tastiera.

```
41A (word) Puntatore all'inizio del buffer di tastiera (head pointer)
41C (word) Puntatore alla fine del buffer di tastiera (tail pointer)
41E (16 word) Il buffer di tastiera
```

Nel buffer di tastiera sono conservati tutti i codici dei tasti introdotti dall'operatore ma non ancora processati dal programma; l'area all'indirizzo 0000:041E è gestita come un buffer circolare, e le word agli indirizzi 0000:041A e 0000:041E puntano rispettivamente alla locazione a cui sarà inserito il prossimo tasto premuto dall'operatore e alla locazione che contiene il prossimo tasto che dovrà essere processato dal programma. Si noti che i valori di entrambi i puntatori sono relativi all'indirizzo di segmento 0040, per cui ad esempio il primo byte del buffer è indicato con 001E (e non 041E); queste locazioni sono descritte più in dettaglio nel capitolo che descrive l'accesso a basso livello alla tastiera.

```
440 (byte) Timer per il motore del dischetto
```

Questo byte contiene un valore che indica per quanti *tick* (18esimi di secondo) dovrà rimanere acceso il motore del dischetto; dopo ogni lettura o scrittura il Dos memorizza in questo byte il valore 37, che forza il motore a rimanere attivo per circa due secondi: in questo modo eventuali accessi successivi allo stesso disco non dovranno attendere che il dischetto raggiunga nuovamente la velocità ottimale. Se un programma prevede di accedere continuamente al dischetto, può modificare questo valore per aumentare il timeout di default; ad esempio, con le istruzioni:

```
DEF SEG = 0
POKE &H440, 255
DEF SEG
```

si forza il controller a lasciare il motore acceso per circa 14 secondi. Se il motore è spento, la modifica di questo byte non sortisce alcun effetto.

`449 (byte) Modo video`

Questo byte contiene un valore che indica il modo video attuale, la stessa informazione restituita dal servizio F dell'interrupt 10h; ovviamente questo valore non può essere modificato.

`44A (byte) Numero di colonne`

Anche in questo caso si tratta di una informazione restituita dal servizio F dell'interrupt 10h, che può essere letta direttamente da un programma, ma non dovrebbe assolutamente essere modificata.

`450 (8 word) Posizione del cursore`

In queste word sono conservate le posizioni dei cursori in ciascuna delle otto pagine testo possibili; il primo byte di ciascuna word indica la colonna (da 0 a 79) e il secondo byte indica la riga (da 0 a 24 oppure 42 o 49, a seconda del numero di righe visualizzate); la posizione del cursore può essere ricavata o anche modificata agendo su queste locazioni, ma tutto sommato è preferibile utilizzare i comandi e le funzioni standard LOCATE, CSRLIN e POS

`460 (2 byte) Aspetto del cursore`

Questi due byte restituiscono la dimensione corrente del cursore: il primo corrisponde alla riga di scansione finale, il secondo alla riga di scansione iniziale; si tratta quindi degli stessi valori impostati con un comando LOCATE

```
LOCATE , , , startLine%, endLine%
```

queste locazioni possono essere lette, ma se si tenta di modificarle non si sortisce alcun effetto.

`462 (byte) Pagina video`

Questo byte restituisce il numero di pagina video attuale, compreso tra 0 e 7.

`463 (word) Indirizzo del controller video`

Questa locazione contiene l'indirizzo della porta del controller 6845 per il video; normalmente conserva il valore esadecimale 3B4 (scheda MDA o Hercules) oppure 3D4 (scheda CGA, EGA e VGA), per cui la lettura di questa word permette di sapere se il computer dispone di una scheda a colori o meno

```
FUNCTION ColorCard%
    '_______________________________
    ' Restituisce -1 se è presente una scheda video a colori
    '_______________________________
    DEF SEG = 0
    ColorCard% = (PEEK(&H463) = &HD4)
```

```
        DEF SEG
    END FUNCTION
465 (doppia word) Orario di sistema
```

Questi quattro byte restituiscono il numero di *tick* trascorsi dalla mezzanotte, e ad esempio sono usati dal BASIC per calcolare il valore delle funzioni TIME$ e TIMER, e dal Dos per impostare la data di modifica dei file.

```
471 (byte) Indicatore di Break
```

Quando il BIOS rilava la pressione del tasto Break sulla tastiera attiva il bit 7 di questa locazione di memoria; in questo modo un programma compilato può reagire al tasto di Break anche se non è stato compilato con l'opzione /D, in questo modo

```
    DEF SEG = 0
    BreakPressed% = (PEEK(&H471) AND 128) <> 0
    DEF SEG
472 (word) Reboot da tastiera
```

Quando l'operatore preme la combinazione Ctrl-Alt-Canc, il BIOS memorizza in questa locazione il valore esadecimale 1234 e poi resetta il sistema, saltando all'indirizzo FFFF:000. Possiamo allora scrivere due procedure BASIC che provocano un reset "a freddo" e "a caldo" dell'intero sistema

```
    SUB ColdReset
        '________________________________
        ' Reset "a freddo" del sistema
        ' equivalente alla pressione del tastino di reset
        '________________________________
        DEF SEG = &HFFFF
        CALL Absolute (0)                ' salta all'indirizzo FFFF:0000
        ' (l'esecuzione non arriva mai a questo punto!)
    END SUB
    SUB WarmReset
        '________________________________
        ' Reset "a caldo" del sistema
        ' equivalente alla pressione di Ctrl-Alt-Canc
        '________________________________
        DEF SEG = 0
        POKE &H472, &H34        ' memorizza il valore 1234
        POKE &H473, &H12        ' nella word a 0000:0472
        DEF SEG = &HFFFF
        CALL Absolute (0)                ' salta all'indirizzo FFFF:0000
        ' (l'esecuzione non arriva mai a questo punto!)
    END SUB
478 (4 byte) Timeout per le stampanti
```

Per ciascuna delle quattro stampanti parallele che è possibile collegare ad un sistema Ms-Dos, questi byte conservano il valore del corrispondente timeout in secondi. Quando un programma tenta di accedere ad una porta parallela a cui non è collegata alcuna stampante oppure è collegata una stampante spenta, il BIOS è in grado di restituire subito il controllo al programma segnalando l'errore; quando invece alla porta parallela è collegata una stampante off-line, il BIOS è costretto ad attendere un certo periodo di tempo per distinguere questa situazione dal caso in cui la stampante è correntemente

impegnata nella stampa di dati ed è solo temporaneamente off-line. Questo intervallo di tempo è detto *timeout della stampante*, ed è impostato al bootstrap con un valore che varia dai pochi secondi fino addirittura ad un paio di minuti. Con la maggior parti delle stampanti più recenti è possibile ridurre il valore di default ad una decina di secondi o meno, senza rischiare di "confondere" il BIOS.

484 (byte) Numero di righe di schermo (solo EGA/VGA)

Questo byte è gestito dal BIOS esteso installato con le schede EGA e VGA, e riporta il valore massimo per la riga del cursore; sapendo che la prima riga dello schermo è la riga zero, possiamo scrivere una semplice funzione che restituisce il numero di righe nel modo video attuale

```
FUNCTION ScreenRows%
        DEF SEG = 0
        ScreenRows% = PEEK(&H484) + 1
        DEF SEG
END FUNCTION
```

487 (byte) Memoria di schermo (solo EGA/VGA)

I bit 5-6 di questo byte conservano la quantità di memoria installata sulle schede EGA e VGA, secondo il seguente schema

```
00      64K                 10      196K
01      128K                11      256K
```

Ecco un breve programma BASIC che calcola questa quantità

```
DEF SEG = 0
videoMem% = ((PEEK(&H487) AND &H60) \ 32 + 1) * 64
DEF SEG

PRINT "La scheda video disponde di "; videoMem%; "K di memoria"
```

4F0 (16 byte) Area di comunicazione tra programmi

Questi 16 byte possono essere usati liberamente per scambiare delle informazioni tra programmi, ad esempio per passare dei dati ad altri programmi lanciati per mezzo di un comando SHELL o RUN; anche se non sono molti i programmi che sfruttano quest'area, è preferibile che i dati siano memorizzati solo un attimo prima di lanciare il programma, o meglio ancora che il programma "ricevente" esegua un checksum sui dati o controlli una "firma" per accertarsi che i valori nell'area siano proprio quelli immessi dal primo programma.

500 (byte) Stato dell'hardcopy dello schermo

Questo byte è gestito dal BIOS per controllare le operazioni di hardcopy dello schermo, e può contenere uno dei seguenti valori:

```
00      OK, pronto ad eseguire l'hardcopy
01      operazione di hardcopy in corso
FF      errore durante l'ultima operazione di hardcopy
```

Anche se questa locazione non dovrebbe essere manipolata dai programmi applicativi, la tentazione di "imbrogliare" il BIOS è troppo forte; ad esempio,

possiamo di fatto disabilitare il tasto PrintScreen con questo semplice comando:

```
DEF SEG = 0
POKE &H500, 1
DEF SEG
```

Ovviamente, il tasto PrintScreen dovrebbe essere riabilitato al termine del programma, altrimenti esso rimarrà inattivo fino al reset successivo.

504 (byte) Unità floppy attiva

Questo byte è utilizzato dal Dos sui sistemi con un unico driver per dischetti, per memorizzare quale dei due driver logici A: o B: è correntemente attivo; se la locazione contiene il valore 0 è attivo il driver A:, se contiene il valore 1 è attivo il driver B:. Questa informazione può essere sfruttata dai programmi applicativi per evitare che il Dos mostri un messaggio all'operatore per richiedere di inserire un dischetto, ad es.

```
' mostra la directory del dischetto
drive$ = "A:"
DEF SEG = 0
IF PEEK(&H504) = 1 THEN drive$ = "B:"
DEF SEG
SHELL "DIR " + drive$
```

Il valore di questo byte può anche essere modificato dal programma con sufficiente sicurezza; quindi il precedente frammento di codice può essere riscritto in questo modo

```
' mostra la directory del dischetto
DEF SEG = 0                    ' assicurati che il Dos
POKE &H504, 0                  ' indirizza il floppy come
DEF SEG                        ' l 'unità A:
SHELL "DIR A:"
```

LE VERSIONI DEL BIOS

Vi sono un paio di informazioni memorizzate direttamente nel ROM BIOS, a cui i programmi BASIC possono accedere per mezzo di alcune PEEK, ma che ovviamente non possono essere modificate.

Il *codice identificativo* del BIOS è un byte che si trova all'indirizzo FFFF:000E e che può essere letto dai programmi per determinare su quale tipo di macchina essi sono eseguiti; purtroppo a partire dagli AT i produttori di BIOS non hanno seguito uno schema affidabile, e molti dei sistemi 386 e 486 in circolazione riportano il medesimo identificativo dei primi AT, pur essendo macchine completamente differenti. Ad ogni modo, ecco una procedura BASIC che legge il byte in questione e stampa un messaggio appropriato

```
SUB ShowBiosVersion
     DEF SEG = &HFFFF
     byte% = PEEK(14)
     DEF SEG
     SELECT CASE byte%
            CASE 255: PRINT "IBM PC"
            CASE 254: PRINT "IBM XT"
            CASE 253: PRINT "IBM PCjr"
            CASE 252: PRINT "IBM AT o superiore"
            CASE ELSE: PRINT "Identificativo sconosciuto"
     END SELECT
END SUB
```

Qualche byte prima dell'identificativo del BIOS è memorizzata la data di rilascio del BIOS stesso, che può essere usata dai programmi a fini diagnostici; questa data è memorizzata negli indirizzi da FFFF:0005 a FFFF:000C nel formato "mm/gg/aa", cioè lo stesso formato restituito dalla funzione DATE$ ma con due sole cifre per l'anno. Ecco una funzione BASIC che preleva il valore e lo restituisce in una stringa

```
FUNCTION BiosDate$
       DIM result$, i%
       DEF SEG = &HFFFF
       FOR i% = 5 to 12
              result$ = result$ + CHR$(PEEK(i%))
       NEXT
       BiosDate$ = result$
END FUNCTION
```

Ad esempio, trovando un risultato anteriore al 19 Ottobre 1981 (data della prima revisione del BIOS) dovreste avvertire l'operatore che vi potrebbero essere alcuni bug, mentre una data anteriore al 16 Agosto 1982 individua un BIOS non in grado di gestire i dischi rigidi; certo, la probabilità di imbattervi in uno di questi reperti storici è bassa, ma non per questo completamente trascurabile.

4

LA TASTIERA

A volte è necessario interagire con la tastiera in modi ben più sofisiticati delle istruzioni che il Basic mette a disposizione. Ad esempio, come è possibile determinare lo stato dei tasti ALT o CTRL ? oppure modificare lo stato dei tasti di bloccaggio CapsLock, NumLock e ScrollLock ? o capire se e quale tasto è stato premuto dall'operatore senza prelevarlo dalla tastiera ? Evidentemente abbiamo bisogno di conoscere in funzionamento "a basso livello" della tastiera, per cui prima di proseguire cercherò di riassumere a grandi linee come funziona la tastiera su un personal computer IBM compatibile.

IL BUFFER DI TASTIERA

Ogni volta che viene premuto o rilasciato un tasto, sia esso un tasto alfanumerico, un tasto cursore o semplicemente un tasto di shift, la circuiteria interna del PC genera un interrupt 09; in realtà esistono delle differenze importanti tra le tastiere avanzate usate attualmente e quelle originariamente montate sui primi PC, ma dal nostro punto di vista possiamo ignorare questo aspetto. Quando avviene un interrupt 09 il controllo è ceduto ad una particolare routine del BIOS, il cui compito è di salvare il contenuto di tutti i registri del microprocessore e processare i segnali in arrivo dal controller della tastiera: l'operazione di salvataggio dei registri (compreso il registro dei flag) è particolarmente importante, in quanto l'utente può premere un tasto in

qualsiasi momento, anche nel mezzo di una copia di file o di unaoperazione di stampa. Dopo aver preso queste precauzioni, il BIOS può analizzare i segnali in arrivo al microprocessore e ricavare il cosiddetto *scan code* del tasto - mediante la seguente sequenza di istruzioni Assembly:

```
IN      AL,60H          ; leggi lo scan code del tasto
TEST    AL,80H          ; testa il bit più significativo
JNZ     Rilasciato      ; se attivo, si tratta di un taso rilasciato
JMP     Premuto         ; altrimenti si tratta di un tasto premuto
                        ; lo scan code è nei bit 0-6 di AL
```

a questo punto il BIOS distingue se si tratta di uno dei tasti di shift (cioè Alt, Ctrl o Shift), di un tasto di lock (CapsLock, NumLock, ScrollLock e il tasto Insert), oppure di un tasto normale; il comportamento è differente a seconda dei casi. Essendo più semplice il trattamento dei tasti di shift e di lock lo analizzeremo per primo.

Una porzione della memoria "bassa" di tutti i PC IBM compatibili è dedicata alla memorizzazione delle variabili del BIOS, ed è spesso detta BIOS *Area*. Quest'area comincia all'indirizzo 0000:0400 e conserva la configurazione del sistema, la posizione del cursore sullo schermo, lo stato del controller dei dischetti e, appunto, alcune informazioni che permettono al BIOS di interpretare le azioni dell'operatore sulla tastiera. In particolare, il byte all'indirizzo 0000:0417 riflette lo stato corrente dei tasti di shift e di lock, secondo il seguente schema:

bit 0 tasto SHIFT destro premuto

bit 1 tasto SHIFT sinistro premuto

bit 2 tasto CTRL premuto

bit 3 tasto ALT premuto

bit 4 stato di SCROLL LOCK

bit 5 stato di NUMLOCK

bit 6 stato di CAPS LOCK

bit 7 stato di INSERT

quando viene premuto o rilasciato un tasto di shift il BIOS non deve fare altro che invertire lo stato del bit corrispondente; nel caso dei tasti di lock il BIOS inverte lo stato del bit corrispondente solo alla pressione del tasto, non al suo rilascio.

Il caso dei tasti "normali" (cioè i tasti alfanumerici e di punteggiatura, ma anche i tasti funzione, i tasti per il movimento del cursore, ecc.) è più complesso, in quanto il BIOS deve considerare lo stato corrente dei tasti di lock (ad es. se CapsLock è premuto o meno) e di shift ("A" è differente da CTRL-

"A" e da SHIFT-"A"). Sebbene siamo abituati a trattare tutti i tasti di lock alla stessa stregua, in realtà vi sono importanti differenze: in particolare, soltanto i tasti CapsLock e NumLock sono trattati direttamente dal BIOS per decodificare lo scan code, mentre la gestione di ScrollLock e Insert è lasciata ai programmi applicativi, che infatti li usano per scopi anche abbastanza differenti.

Dopo aver decodificato il tasto premuto il Bios genera un *codice* ASCII *esteso*, lungo due byte nel seguente formato:

- byte meno significativo: il codice ASCII dei tasti normali, oppure zero per i tasti del set esteso
- byte più significativo: zero per i tasti normali, oppure il codice di scansione per i tasti del set esteso

dove per tasti "normali" si intende il set ASCII 0-255 e quindi tutti i caratteri visualizzabili più le combinazioni CTRL-tasto, mentre il set esteso comprende i tasti cursore, i tasti funzione, le combinazioni ALT+tasto, ecc. Ad esempio, il codice ASCII esteso del tasto "A" è (65,0), mentre il codice del tasto funzione F1 è (0,59). Il BIOS memorizza il codice così calcolato nel *buffer di tastiera*, un'area lunga 32 byte che si trova anch'essa nella BIOS A*rea* all'indirizzo 0000:041E. Il motivo per cui viene usato un buffer anziché una singola word è semplice: in questo modo il BIOS può continuare ad accumulare caratteri mentre l'operatore li digita alla tastiera, anche se il programma non è ancora in grado di processarli. Molti utenti sfruttano inconsapevolmente questa caratteristica quando incominciano a digitare un comando Dos prima che sia terminata l'esecuzione del comando precedente.

Il buffer di tastiera è organizzato come un *buffer circolare*, nel senso che il BIOS riempie tutte le locazioni fino a 0000:043D, poi riprende dalla locazione 0000:041E; ovviamente, per scrivere nuovamente in 0000:041E è necessario che nel frattempo il programma applicativo attivo in quel momento abbia cominciato a leggere i tasti premuti in precedenza. Per tenere traccia dello stato corrente del buffer il BIOS gestisce due locazioni di memoria che fungono da puntatori, e più precisamente

0040:001A puntatore al prossimo codice che sarà inviato al programma (*head pointer*)

0040:001C puntatore alla locazione in cui inserire il codice letto dalla tastiera (*tail pointer*)

0040:001E inizio del buffer di tastiera

0040:0043 fine del buffer di tastiera

(Notate che 0040:001E *è un altro modo di scrivere* 0000:041E); *ovviamente* head pointer *significa "puntatore di testa" mentre* tail punter *significa "puntatore di coda".*

Riassumendo, per ogni tasto premuto dall'operatore il BIOS calcola il codice ASCII esteso e lo memorizza alla locazione il cui indirizzo (relativo al segmento 0040) è conservato nel tail pointer, poi incrementa quest'ultimo di due e restituisce il controllo al programma attivo in quel momento. Quando il programma richiede un tasto (mediante l'interrupt 16H, come vedremo tra un attimo), il BIOS recupera il codice memorizzato alla locazione indicata dall'head pointer, lo passa al programma e incrementa il puntatore di due. Ovviamente, quando uno dei due puntatori arriva a puntare oltre la fine del buffer, il suo valore viene "normalizzato" per farlo puntare all'inizio. Alla richiesta del programma il BIOS deve controllare se vi sono tasti nel buffer (verificando che l'head pointer e il tail pointer abbiano valori differenti); in modo simile, prima di inserire un nuovo scan code nel buffer la routine che serve l'interrpt 09 deve accertarsi che il tail pointer non punti alla locazione immediatamente precedente a quella indicata dall'head pointer, in quanto il successivo incremento porterebbe i due puntatori allo stesso valore, che però - come abbiamo appena detto - è la condizione di "buffer vuoto". In definitiva, nel buffer di 32 byte vi è posto per soli 15 tasti, e non sedici come sarebbe lecito aspettarsi.

I SERVIZI DEL BIOS

Vediamo cosa accade sul versante del programma applicativo che deve leggere la tastiera, e in particolare di un programma scritto in BASIC. Qualunque sia l'istruzione utilizzata dal programma - INKEY$, INPUT$, INPUT o LINE INPUT - internamente tutte le richieste al BIOS avvengono per mezzo dell'interrupt 16H, e in particolare per mezzo dei servizi 0 e 1. Ecco cosa farebbe un programmatore Assembly per richiedere l'input di un carattere mediante il servizio 0:

```
MOV     AH,0            ; numero del servizio in AH
INT     16H             ; richiama il Bios
                        ; restituisce il tasto premuto in AX
CMP     AH,0            ; se AH=0 si tratta di un tasto normale
JZ      Norm            ; altrimenti si tratta di un tasto esteso
```

Ecco la procedura equivalente scritta in BASIC; per comodità i tasti normali sono restituiti come numeri positivi (ad es. 65 per "A"), mentre per i tasti del set esteso (tasti cursore, tasti funzione, ecc.) è restituito il codice di scansione cambiato di segno (ad es. -59 per F1):

```
FUNCTION FetchKey% ()
	'___________________________
	' restituisce il codice di scansione del tasto premuto
	'___________________________
	DIM reg AS RegType
	reg.ax = 0                      ' servizio zero in AH
	Interrupt &H16, reg, reg ' richiama il Bios
	IF reg.ax AND 255 THEN
		FetchKey% = reg.ax AND 255                ' se AL<>0 il tasto è in AL
	ELSE
		FetchKey% = -((reg.ax AND &HFF00&) \ 256)
	END IF
END FUNCTION
```

Il difetto di questo servizio è di interrompere l'elaborazione del programma fino a quando l'utente non preme un tasto, un po' come l'istruzione PAUSE del Dos, per cui non può essere utilizzato nelle situazioni in cui si desidera eseguire il *polling* della tastiera, come ad esempio nel seguente frammento di programma:

```
	DO
		LOCATE 1, 1: PRINT TIME$;
	LOOP UNTIL INKEY$ <> ""
```

In questi casi è preferibile utilizzare il servizio 1 dell'interrupt 16H, che testa la presenza di un tasto nel buffer di tastiera e ne restituisce il valore in AX, ma non lo rimuove dal buffer stesso. Ecco come userebbe questo servizio un programmatore Assembly:

```
	MOV	AH,1		; numero del servizio in AH
	INT	16H		; richiama il Bios
	JNZ	Premuto		; se il flag zero è disattivo è stato
				; premuto un tasto (il suo valore è in AX)
```

Se decidiamo di scavalcare il BASIC e leggere i tasti direttamente dal BIOS siamo in grado di effettuare una serie di "trucchi" davvero notevoli. Ad esempio, possiamo costruire una funzione PrefetchKey in grado di leggere il codice di un tasto senza prelevarlo realmente dal buffer:

```
FUNCTION PrefetchKey% ()
	'___________________________
	' restituisce il codice del tasto premuto oppure zero
	' ma non rimuove il tasto dal buffer
	'___________________________
	DIM reg AS RegType
	reg.ax = &H100   ' servizio uno in AH
	Interrupt &H16, reg, reg        ' richiama il Bios
	IF reg.flags AND 64 THEN        ' testa il flag di zero (bit 6)
		PreFetchKey% = 0        ' nessun tasto se attivato
	ELSEIF reg.ax AND 255 THEN
		PreFetchKey% = reg.ax AND 255     ' se AL<>0 il tasto è in AL
	ELSE
		PreFetchKey% = -((reg.ax AND &HFF00&) \ 256)
	END IF
END FUNCTION
```

Ecco come sfruttare la routine in un programma:

```
' esegui una INPUT se il primo tasto è una cifra 0-9
' altrimenti esegui una istruzione LINE INPUT
' ignora i tasti cursore, i tasti funzione, ecc.
DO
        ky = PrefetchKey
LOOP UNTIL ky > 0
IF ky>=ASC("0") AND ky<=ASC("9") THEN
        INPUT numero
ELSE
        LINE INPUT stringa$
END IF
' si noti che il controllo     (ky AND &HFF00) = 0
' serve ad accertarsi che AH sia zero, e che quindi
' abbiamo a che fare con un tasto normale e non un tasto
' funzione o per il movimento del cursore.
```

L'uso delle funzioni FetchKey e PrefetchKey in luogo della INKEY$ ha anche un altro vantaggio, quello di aggirare un fastidioso bug del BASIC descritto in un'altro capitolo di questo testo.

I TASTI DI SHIFT E DI LOCK

Il servizio 02 dell'interrupt 16H restituisce in AL il valore del byte all'indirizzo 0040:0017 e permette quindi di conoscere lo stato dei vari tasti di shift e di lock. Il seguente listato riporta un esempio di programma che mostra lo stato di tutti i tasti di questo tipo; ovviamente lo stato del byte in questione può essere ottenuto altrettanto facilmente con una istruzione PEEK, ma se è possibile ottenere un risultato usando un metodo "educato" (well-behaved) e ben documentato è inutile e potenzialmente pericoloso fare altrimenti.

```
'_________________________
' mostra lo stato dei tasti di shift e di lock
'_________________________

DIM reg AS RegType
reg.ax = &H200                ' servizio due, in AX
Interrupt &H16, reg, reg      ' chiama il Bios
                              ' restituisce il byte in AL
IF reg.ax AND 1 THEN
      PRINT "Tasto SHIFT destro premuto"
END IF
IF reg.ax AND 2 THEN
      PRINT "Tasto SHIFT sinistro premuto"
END IF
IF reg.ax AND 4 THEN
      PRINT "Tasto CTRL premuto"
END IF
IF reg.ax AND 8 THEN
      PRINT "Tasto ALT premuto"
END IF
IF reg.ax AND 16 THEN
```

```
        PRINT "Tasto SCROLL LOCK attivato"
END IF
IF reg.ax AND 32 THEN
        PRINT "Tasto NUM LOCK attivato"
END IF
IF reg.ax AND 64 THEN
        PRINT "Tasto CAPS LOCK attivato"
END IF
IF reg.ax AND 128 THEN
        PRINT "Tasto INSERT attivato"
END IF
```

Se però intendiamo modificare uno o più bit in questo byte dobbiamo forzatamente ricorrere alla manipolazione diretta della memoria con una istruzione POKE, in quanto il BIOS non mette a disposizione alcun servizio per tale scopo. Il seguente listato riporta alcune brevi subroutine che attivano e disattivano i tasti CapsLock e NumLock; è facile modificarle per agire sui tasti ScrollLock e Insert.

```
SUB CapsLockOn ()
        DEF SEG = &H40
        POKE &H17, PEEK(&H17) OR &H40
        DEF SEG
END SUB


SUB CapsLockOff ()
        DEF SEG = &H40
        POKE &H17, PEEK(&H17) AND NOT &H40
        DEF SEG
END SUB


SUB NumLockOn ()
        DEF SEG = &H40
        POKE &H17, PEEK(&H17) OR &H20
        DEF SEG
END SUB


SUB NumLockOff ()
        DEF SEG = &H40
        POKE &H17, PEEK(&H17) AND NOT &H20
        DEF SEG
END SUB
```

Sebbene queste routine abbiano effetto immediato, lo stato dei led sulla tastiera sarà aggiornato soltanto alla successiva invocazione dell'interrupt 09, ossia alla successiva pressione o rilascio di un qualsiasi tasto. Non provate ad usare questo sistema per agire sui tasti SHIFT, CTRL e ALT, in alcuni casi potreste provocare il blocco del sistema.

OPERAZIONI SUL BUFFER DI TASTIERA

Manipolando direttamente il buffer di tastiera si possono ottenere dei risultati davvero interessanti. Ad esempio, in alcuni casi è necessario svuotare il buffer per essere sicuri che l'utente non batta accidentalmente un tasto prima di leggere una domanda e prima che il BASIC sia pronto ad accettare la risposta. Potremmo scriveremmo qualcosa del genere:

```
DO: LOOP WHILE INKEY$ <> ""
PRINT "Vuoi formattare il dischetto (S/N) ?"
```

ma poiché sappiamo che il buffer di tastiera è "vuoto" quando l'head pointer e il tailpointer hanno lo stesso valore, possiamo ottenere lo stesso risultato scrivendo:

```
SUB ClearKeyboardBuffer
      DEF SEG = &H40
      POKE &H1C, PEEK(&H1A)
      POKE &H1D, PEEK(&H1B)
      DEF SEG
END SUB
```

ad essere pignoli, esiste una probabilità (molto piccola) che proprio nel mezzo di questa sequenza di istruzioni capiti un interrupt 9 e confonda le idee al BIOS. Per rendere questo sistema infallibile dovremmo disattivare gli interrupt, cosa che però non è possibile in BASICe richiede invece una routine in Assembly come la seguente:

```
; pulisce il buffer di tastiera
MOV     AX,40H
MOV     ES,AX               ; ES punta all'area Bios
CLI                         ; disattiva gli interrupt
MOV     AX,ES:[1AH]         ; questo è il valore dell'head pointer
MOV     ES:[1CH]            ; memorizza nel tail pointer
STI                         ; riattiva gli interrupt
```

Un altro trucco abbastanza interessante consiste nel manipolare il buffer e i suoi puntatori per inserire fino a 15 caratteri; attenzione al fatto che stiamo manipolando una delle aree più delicate del computer, e un minimo sbaglio può mandare in tilt il sistema: abbiate quindi un po' di cura a copiare correttamente la seguente routine:

```
SUB InsertKeys (keys$)
      '------------------------------
      ' inserisce fino a 15 caratteri nel buffer di tastiera
      ' non possono essere inseriti tasti funzione, tasti cursore, ecc.
      '------------------------------
      DIM length%, i%
      DEF SEG = &H40
      ' limita la lunghezza di keys$
      length% = LEN(keys$)
      IF length% > 15 THEN length% = 15
      ' inserisci i tasti nel buffer; ad ogni carattere
      ' corrisponde una coppia di byte, il secondo dei
      ' quali è sempre zero
      FOR i% = 0 TO length% - 1
```

```
                POKE &H1E + i% * 2, ASC(MID$(keys$, i% + 1, 1))
                POKE &H1E + i% * 2 + 1, 0
        NEXT
        ' componi il valore 001E nell'head pointer
        POKE &H1A, &H1E
        POKE &H1B, 0
        ' componi il valore del tailpointer
        POKE &H1C, &H1E + length * 2
        POKE &H1D, 0
        DEF SEG
END SUB
```

Si noti che, così com'è, la procedura InsertKeys permette di inserire soltanto i tasti del set ASCII standard, e non i tasti funzione, i tasti cursore, ecc.; capito il meccanismo questa modifica non dovrebbe essere troppo complessa. Ecco come è possibile sfruttare questa procedura in un programma BASIC:

```
' richiama INPUT suggerendo una risposta all'operatore
PRINT "Inserisci l'orario corrente: ";
InsertKeys TIME$
LINE INPUT orario$
```

Questa tecnica può anche essere utile nella fase di debug di un programma, per evitare di dover ogni volta rispondere a tutte le domande prima di arrivare al punto che vogliamo testare; useremo allora istruzioni di questo tipo:

```
PRINT "Inserisci l'orario corrente: ";
IF debugMode THEN InsertKeys TIME$ + CHR$(13)
LINE INPUT orario$
```

dove CHR$(13) simula la pressione del tasto INVIO e di fatto evita che il programma si blocchi sull'istruzione LINE INPUT in attesa della risposta dell'operatore. Attenzione a non lasciare dei tasti nel buffer quando il controllo ritorna all'ambiente di sviluppo, in seguito ad un breakpoint oppure ad un comando STOP o END; se necessario, svuotate il buffer con uno dei metodi visti in precedenza.

```
' pulisci il buffer prima di un breakpoint
DO: LOOP UNTIL INKEY$ = ""
STOP
```

La procedura InsertKeys è anche molto utile per inviare dei comandi ad un programma esterno richiamato mediante SHELL; il limite dei quindici caratteri pone una restrizione pesante alla applicazione pratica di questo metodo, non di meno essa può essere sfruttata con notevole efficacia: ad esempio, è possibile richiamare un word processor e fargli stampare un documento prima di tornare al programma BASIC. Purtroppo, alcuni programmi hanno la deprecabile abitudine di pulire il buffer di tastiera, rendendo inutili i nostri sforzi: un esempio sono tutti i comandi Dos potenzialmente pericolosi, come FORMAT o DISKCOPY.

Si tenga inoltre presente che, per quanto detto prima, questa routine potrebbe fallire a causa di una chiamata all'interrupt 09 nel mezzo dell'esecuzione, evento che stavolta è leggermente più probabile in quanto le istruzioni eseguite sono numerose e anche relativamente lente (come la funzione MID$).

Se siete sicuri che il vostro programma sarà eseguito soltanto su su sistemi dotati di tastiera estesa, potete sfruttare il servizio 5 dell'interrupt 16h (introdotto sui computer 80286 o superiori), che evita la manipolazione diretta del buffer e non crea conflitti con l'interrupt 09

```
SUB InsertKeys2 (keys$)
    '___________________________________
    ' inserisce fino a 15 caratteri nel buffer di tastiera
    ' non possono essere inseriti tasti funzione, tasti cursore, ecc.
    ' richiede computer con tastiera estesa
    '___________________________________
    DIM reg AS RegType, i%, length%
    ' limita la lunghezza di keys$
    length% = LEN(keys$)
    IF length% > 15 THEN length% = 15
    ' inserisci i tasti nel buffer; ad ogni carattere
    ' corrisponde una coppia di byte, il secondo dei
    ' quali è sempre zero
    FOR i% = 0 TO length% - 1
        reg.ax = &H500   ' servizio 5
        reg.cx = ASC(MID$(keys$, i% + 1)) ' codice in CL
        Interrupt &H16, reg, reg
    NEXT
END SUB
```

Infine, si tenga presente che non tutte le applicazioni commerciali leggono la tastiera mediante l'interrupt 16: ve ne sono alcune, infatti, che prendono il controllo dell'interrupt 09 e processano i tasti direttamente, affiancandosi o in alcuni casi sostituendosi al BIOS; in tal caso le procedure **InsertKeys** e **InsertKeys2** non sortiscono alcun effetto

IL "TYPEMETIC RATE"

Il servizio 03 dell'interrupt 16h permette di impostare il cosiddetto *typematic rate*, ossia la velocità con cui il controller della tastiera attiva la ripetizione dei caratteri quando si tiene premuto un tasto, e il ritardo con cui appare la prima ripetizione. La procedura **SetTypemeticRate** esegue tutte le operazioni necessarie:

```
SUB SetTypematicRate (velocita%, ritardo%)
    '_________________________
    ' imposta il "typematic rate"
    '_________________________
    DIM reg AS RegType
    reg.ax = &H305                      ' servizio 3, sottofunzione 5
    reg.bx = ritardo% * 256 + velocita% ' ritardo in BH, velocità in BL
    Interrupt &H16, reg, reg            ' richiama il Bios
END SUB
```

mentre la tavola seguente riporta tutti i possibili valori per i parametri **velocita%** e **ritardo%:**

RITARDO

0	250 millisecondi
1	500 "
2	750 "
3	1000 "

VELOCITA'

0	30.0 caratteri/secondo
1	26.7 " "
2	24.0 " "
3	21.8 " "
4	20.0 " "
5	18.5 " "
6	17.1 " "
7	16.0 " "
8	15.0 " "
9	13.3 " "
10	12.0 " "
11	10.9 " "
12	10.0 " "
13	9.2 " "
14	8.6 " "
15	8.0 " "
16	7.7 " "
17	6.7 " "
18	6.0 " "
19	5.5 " "
20	5.0 " "
21	4.6 " "
22	4.3 " "
23	4.0 " "
24	3.7 " "
25	3.3 " "
26	3.0 " "
27	2.7 " "
28	2.5 " "
29	2.3 " "
30	2.1 " "
31	2.0 " "

Se il vostro programma richiede spesso di tenere premuto un tasto, ad es. per scrollare il contenuto di una finestra, può essere molto utile usare questa procedura per aumentare la velocità di ripetizione. Attenzione a non esagerare però: una velocità troppo alta (o un ritardo troppo breve) rende difficile il controllo del cursore.

5

IL MOUSE

Non perderò tempo a spiegare cosa è il mouse e come funziona, se non per ricordare che esso può essere installato direttamente in CONFIG.SYS, mediante una istruzione DEVICE, o lanciando un programma TSR da AUTOEXEC.BAT oppure dal prompt del Dos. Sebbene si tratti di opzioni completamente diverse, dal punto di vista dei programmi applicativi esse producono il medesimo risultato, e l'una o l'altra può essere usata indifferentemente a seconda delle necessità e dei gusti. Si tenga presente, comunque, che la versione come device driver consuma un po' meno memoria della versione installata come TSR.

A differenza di alcune altre periferiche, l'interfaccia software offerta dai mouse prodotti dai vari produttori (quella che in gergo si suole chiamare API, o *Application Programming Interface*) è standard, e si basa su una serie di servizi offerti ai programmi applicativi per mezzo dell'interrupt 33 esadecimale. In questi tempi sembra perfettamente normale che i criteri per utilizzare una periferica così diffusa come il mouse siano uno "standard", ma fino a non molti anni fa ciò non era affatto scontato. Infatti, i primi fabbricanti di mouse erano così ansiosi di proporre il *miglior* mouse in commercio, che spesso aggiungevano funzioni nuove e più potenti, che però nella maggior parte dei casi potevano essere sfruttate da un numero davvero esiguo di applicazioni (spesso solo quelle fornito a corredo del mouse stesso), e che in fin dei conti servivano solo a creare problemi di compatibilità. Un esempio delle conseguenze di questa situazione è - ad esempio - il fatto che a tutt'oggi nessun applicativo sfrutta il bottone centrale del mouse, proprio perché non tutti i

mouse sono a tre tasti. Uno dei meriti della Microsoft - che oltre ad essere la più grande software house del mondo è anche uno dei maggiori produttori di mouse - è proprio quello di aver imposto uno standard, a cui tutti gli altri si sono dovuti adeguare, soprattutto grazie alla onnipresenza di applicativi quali MS Word, MS Works, i vari compilatori Quick Basic e Quick C, per non parlare di Windows. In BASIC il dialogo con il driver del mouse si realizza per mezzo delle istruzioni INTERRUPT e INTERRUPTX. Piuttosto che infarcire i nostri programmi con questi comandi, come in altre occasioni seguiremo un approccio "modulare" e creeremo delle routine che accedono al mouse, a tutto vantaggio della leggibilità del codice. Per accedere ai servizi del driver occorre caricare il numero del servizio in AX, memorizzare eventualmente altri valori nei registri BX, CX e DX, e richiamare l'interrupt 33h. Si noti che tranne pochissime eccezioni, i registri di segmento ES e DS non sono utilizzati, per cui l'istruzione INTERRUPTX non sarà quasi mai necessaria.

TEST DEL DEVICE DRIVER

Ovviamente, la prima cosa da fare in un programma che intende sfruttare il mouse è proprio controllare che il mouse sia installato e perfettamente funzionante. Il *Technical Reference* del mouse ci informa che per testare la presenza del sorcetto è sufficiente richiamare il servizio 0 e testare il valore restituito in AX, in questo modo:

```
reg.ax = 0
INTERRUPT &H33, reg, reg
IF reg.ax = -1 THEN PRINT "Mouse installato"
```

Il ragionamento alla base di questo metodo è il seguente: se il driver è effettivamente installato in memoria, l'attivazione dell'interrupt 33h gli cederà il controllo e gli darà il modo di modificare il registro AX con un valore diverso da zero, in modo da segnalare al programma applicativo la presenza del driver stesso. Se invece il driver non è installato il controllo ritorna immediatamente al programma, con ancora il valore zero in AX. Purtroppo con i computer le cose non sono mai semplici come sembrano e quello che il *Technical Reference* e i duemilacinquecento libri copiati pedissequamente da quest'ultimo dimenticano di dire è che non tutti i BIOS, al bootstrap, inizializzano correttamente i vettori di interrupt per farli puntare ad una istruzione IRET (Return From Interrupt); in particolare, molti dei primi BIOS inizializzavano solo i alcuni dei vettori, e azzeravano tutti gli altri. Questo significa che su alcuni vecchi PC sprovvisti di mouse, delle istruzioni come quelle appena viste tentavano di eseguire una routine all'indirizzo 0000:0000, ma a questo indirizzo non vi sono istruzioni eseguibili; il risultato, inutile dirlo, è un bel crash di sistema !!!

Fortunatamente esiste un rimedio abbastanza semplice, che è quello adottato dalla stessa Microsoft (che di queste cose se ne intende...), come si può verificare studiando il sorgente della User Interface Library inclusa nel BASIC PDS. È sufficiente accertarsi che il vettore dell'interrupt 33h non sia una doppia word nulla:

```
' leggi il vettore dell'interrupt 33h
DEF SEG = 0
offset& = PEEK(&H33 * 4) + 256& * PEEK(&H33 * 4 + 1)
segment& = PEEK(&H33 * 4 + 2) + 256& * PEEK(&H33 * 4 + 3)
DEF SEG
' se il vettore è nullo il mouse non è installato
IF offset& = 0 AND segment& = 0 THEN GOTO MouseNonTrovato
```

Oltre a testare la presenza del mouse, il servizio 0 resetta lo stato interno del driver e restituisce il numero dei bottoni del mouse nel registro BX. La procedura **MouseInit** unifica quanto detto finora e può essere richiamata dai programmi applicativi all'inizio dell'esecuzione:

```
SUB MouseInit (buttons%)
  '___________________________
  ' Testa la presenza del mouse ed inizializza il driver
  ' restituisce in buttons% il numero dei bottoni,
  ' oppure zero se non è stato trovato il driver
  '___________________________
  DIM offset&, segment&, reg AS RegType
  buttons% = 0
  ' leggi il vettore dell'interrupt 33h
  DEF SEG = 0
  offset& = PEEK(&H33 * 4) + 256& * PEEK(&H33 * 4 + 1)
  segment& = PEEK(&H33 * 4 + 2) + 256& * PEEK(&H33 * 4 + 3)
  DEF SEG
  ' se il vettore è nullo il mouse non è installato
  IF offset& OR segment& THEN
    ' reg.ax = 0
    Interrupt &H33, reg, reg
    IF reg.ax = -1 THEN buttons% = reg.bx
  END IF
END SUB
```

notate come è stato semplificato il test sulla coppia (offset, segment) e che l'istruzione:

```
' reg.ax = 0
```

è stata commentata in quanto, essendo reg una variabile dinamica definita all'inizio della procedura, il valore delle sue componenti è sicuramente zero. Con accorgimenti di questo tipo si riesce a guadagnare byte preziosi e a velocizzare i programmi.

È opportuno spiegare in dettaglio cosa fa il servizio AX = 0; dopo una chiamata a questa funzione il driver si trova nel seguente stato:

rif SIMBOLO 183 \f "Symbol" \s 10 \h il puntatore del mouse è situato al centro dello schermo (pagina zero), ma non è visibile

rif SIMBOLO 183 \f "Symbol" \s 10 \h il cursore ha la forma del "blocchetto" trasparente nei modi testo, oppure della classica freccia nei modi grafici

rif SIMBOLO 183 \f "Symbol" \s 10 \h i movimenti del mouse sono limitati dal numero di righe e colonne del modo testo corrente

rif SIMBOLO 183 \f "Symbol" \s 10 \h il rapporto mickey/pixel è di 8 a 8 in orizzontale, e di 16 a 8 in verticale

rif SIMBOLO 183 \f "Symbol" \s 10 \h la soglia per attivare la doppia velocità è impostata a 64 mickey/secondo

Qualche chiarimento non guasta: il mickey è l'unità di spostamento del mouse, ossia la distanza minima che deve percorrere il mouse per avvertire il movimento, e che equivale circa a 1/200 di pollice (poco più di un decimo di millimetro). Per default il mouse deve essere spostato di 8 mickey per spostare il cursore di 8 pixel a destra o sinistra (quindi un mickey per pixel), e di 16 mickey per spostare il cursore di 8 pixel in alto o in basso (quindi due mickey per pixel). La doppia velocità è invece una caratteristica che hanno i mouse più evoluti, i cosiddetti mouse "balistici", che permette di dimezzare il rapporto mickey/pixel quando il mouse raggiunge una certa velocità, che per default è 64 mickey al secondo. Più avanti vedremo come modificare queste impostazioni di default.

È importante sottolineare un altro aspetto del servizio zero: poiché non si limita a restituire informazioni sul mouse, ed effettua un vero e proprio reset interno del driver, esso dovrebbe essere richiamato una volta soltanto all'inizio del programma, magari conservando il risultato in una variabile globale per testarla poi in seguito. Diversamente, dopo ogni chiamata sarebbe necessario impostare nuovamente tutti i parametri del driver.

Prima di esaminare gli altri servizi del driver, voglio attirare la vostra attenzione su un particolare che può passare inosservato: il servizio AX=0 imposta una finestra di visibilità pari al numero di righe e di colonne presenti sul video *in quel momento*; in altre parole, se dopo aver inizializzato il mouse cambiate il modo video, ad esempio usate l'istruzione WIDTH per attivare il modo testo a 43 o 50 righe, il cursore del mouse diventerà invisibile al di sotto della 25a riga ! Attenzione, quindi: (a) inizializzate il mouse dopo aver impostato il modo video utilizzato dal programma, oppure (b) re-inizializzate il mouse dopo ogni modifica al modo video. Inoltre esistono alcuni driver (meno recenti) il cui servizio zero reimposta sempre la visibilità fino alla 25a riga, indipendentemente dal numero di righe attualmente mostrare sullo schermo.

IL CURSORE DEL MOUSE

Il servizio zero dell'interrupt 33h non rende automaticamente visibile il cursore del mouse, anzi ne forza la disattivazione. Pertanto il cursore deve essere reso visibile mediante una chiamata al servizio AX=1, e può essere reso nuovamente invisibile con il servizio AX=2. Per l'occasione creeremo due procedure BASIC dai nomi più comprensibili **MouseShow** e **MouseHide** :

```
SUB MouseShow
  '________________________
  '  Mostra il cursore del mouse
  '________________________
  DIM reg AS regType
  reg.ax = 1
  Interrupt &H33, reg, reg
END SUB
SUB MouseHide
  '________________________
  '  Nascondi il cursore del mouse
  '________________________
  DIM reg AS regType
  reg.ax = 2
  Interrupt &H33, reg, reg
END SUB
```

Il controllo dello stato del cursore è più importante di quello che potrebbe sembrare a prima vista; infatti, sembrerebbe logico attivare il cursore del mouse una volta soltanto all'inizio del programma, e lasciarlo attivato per tutta la durata dell'esecuzione. Accade però che le istruzioni di I/O su schermo del BASIC, sia in modo testo che in grafica, non prevedono che vi sia un mouse, per cui ad esempio il comando CLS pulisce *tutto* lo schermo, compreso quindi il cursore del mouse. Sembrerebbe allora necessaria una sequenza di questo tipo:

```
CLS
MouseShow
```

che però non funziona ! La sequenza corretta è invece la seguente:

```
MouseHide
CLS
MouseShow
```

In altre parole, occorre disabilitare il mouse prima di ogni istruzione di output, e riattivarlo subito dopo. Lo stesso discorso vale quindi per i comandi PRINT, FILES, DRAW, LINE, CIRCLE, PSET e così via. Un po' meno intuitivo è il motivo per cui il mouse deve essere disabilitato prima della funzione SCREEN (a due e a tre argomenti):

```
MouseHide
asciiCode% = SCREEN(1, 1)
colorCode% = SCREEN(1, 1, 2)
MouseShow
```

in questo caso la sequenza vista ci assicura che questa funzione restituisca informazioni sul contenuto dello schermo e non sul cursore del mouse, che

potrebbe occupare casualmente proprio la posizione testata dal programma in quel momento. Non vi sono, invece, problemi con l'istruzione LOCATE, in quanto il cursore hardware in modo testo e il cursore del mouse non si disturbano a vicenda.

Per usare queste istruzioni occorre fare attenzione ad un altro punto; la procedura **MouseShow** funziona correttamente solo se è stata preceduta da una sola istruzione **MouseHide**. Se ad esempio consideriamo il seguente frammento di codice:

```
MouseHide
MouseHide
CLS
MouseShow
```

il cursore continuerà ad essere invisibile anche dopo l'ultima istruzione ! Per capire il motivo di questo strano comportamento occorre sapere che il driver mantiene internamente un contatore, che è inizializzato a -1 dal servizio AX=0 (**MouseInit**), ed è incrementato ad ogni esecuzione del servizio AX=1 (**MouseShow**) e decrementato ad ogni esecuzione del servizio AX=2 (**MouseHide**). Quando il contatore è nullo il cursore è visibile, altrimenti è invisibile; questo spiega perché al termine del programma precedente il cursore non è ridiventato visibile:

```
' se il mouse è visibile il contatore è = 0
MouseHide    ' contatore = -1
MouseHide    ' contatore = -2
CLS
MouseShow    ' contatore = -1, mouse invisibile!
```

Se credete che sia tutto chiaro, provate ad eseguire il seguente programma:

```
' supponiamo di partire con il mouse visibile...
MouseShow    ' contatore = +1 (???)
MouseHide    ' contatore = 0
```

per quanto possa sembrare paradossale, al termine delle istruzioni precedenti il mouse NON è visibile. Questo si spiega col fatto che **MouseShow** incrementa il contatore interno *solo se il contatore stesso ha un valore negativo*, e non incrementa mai oltre il valore zero. In altre parole, una qualsiasi serie di istruzioni **MouseShow** non porterà mai il contatore oltre il valore zero, e basterà una singola istruzione **MouseHide** per fare scomparire il cursore!

Per quanto possa sembrare artificio e innaturale, a ben vedere il sistema adottato da Microsoft, e poi diventato lo standard, ha numerosi vantaggi. Ad esempio, supponiamo di avere una procedura che racchiude tutte le routine di output su schermo tra coppie di **MouseHide/MouseShow** in questo modo:

```
MouseHide
' istruzioni di output
' .....
MouseShow
```

Una simile procedura funziona perfettamente sia quando il mouse è correntemente visibile (il programma principale ha eseguito una istruzione

MouseShow) sia in caso contrario; infatti, se la procedura inizia con il mouse invisibile l'istruzione **MouseShow** finale non riesce a rendere visibile il cursore del mouse e il mouse continuerà ad essere invisibile. In generale, quindi, è sufficiente aggiungere o eliminare una istruzione **MouseShow** all'inizio del programma per abilitare o disabilitare il mouse in tutta la applicazione, senza dover modificare le singole procedure. Questo è utile, ad esempio, se si vuole dare la possibilità all'utente di disattivare il mouse specificando uno switch sulla riga di comando:

```
MouseInit bottoni%
IF INSTR(COMMAND$, "/NOMOUSE") THEN MouseHide
```

IMPOSTARE LA POSIZIONE DEL CURSORE

Il driver lavora sempre in termini di pixel, anche nel caso in cui lo schermo si trova in modo testo. Per i programmi che non usano la grafica si rende quindi necessaria una operazione di conversione dalle coordinate in pixel alle coordinate in caratteri, o viceversa a seconda del tipo di servizio richiesto al driver. Questa operazione di conversione è abbastanza semplice, tenuto conto che un carattere è pari a otto pixel e che l'angolo in alto a sinistra ha le coordinate (0,0) nel sistema dei pixel e (1,1) in modo testo. Si ha quindi che per convertire le coordinate testo in pixel:

```
X = (COLONNA - 1) * 8
Y = (RIGA - 1) * 8
```

per convertire le coordinate pixel in testo:

```
RIGA = INT (Y / 8) + 1
COLONNA = (X / 8) + 1
```

A questo punto è semplice usare il servizio AX=4 del driver, per impostare la posizione del cursore del mouse in modo testo. Il listato seguente riporta due procedure, una per il modo testo ed una per il modo grafico; notate che la procedura **MouseSetPos** accetta come primo argomento il numero di riga (quindi la coordinata verticale), mentre **MouseSetPosG** richiede come primo argomento il valore della ascissa X (quindi la coordinata orizzontale):

```
' imposta la posizione del cursore (modo testo)
SUB MouseSetPos (riga%, colonna%)
  '---------------------------------
  ' Imposta la posizione del mouse (modo testo)
  '---------------------------------
  DIM reg as RegType
  reg.cx = (colonna% - 1) * 8
  reg.dx = (riga% - 1) * 8
  reg.ax = 4
  Interrupt &H33, reg, reg
END SUB
SUB MouseSetPosG (x%, y%)
```

```
    '-------------------------------
    ' Imposta la posizione del mouse (modo grafico)
    '-------------------------------
    DIM reg as RegType
    reg.cx = x%
    reg.dx = y%
    reg.ax = 4
    Interrupt &H33, reg, reg
END SUB
```

In realtà la procedura **MouseSetPos** non sarà molto usata, in quanto il mouse dovrebbe essere controllato dall'utente, e non dal programma; se la posizione del cursore variasse in modo inaspettato nel corso dell'esecuzione si potrebbe creare un disorientamento dell'operatore. In un caso però questa procedura risulta molto utile, e cioè subito dopo l'inizializzazione del mouse, dato che ho spiegato in precedenza che il serivzio AX=0 posiziona il cursore al centro dello schermo, mentre molti programmi sono soliti spostare il cursore nell'angolo superiore sinistro. Scriveremo allora qualcosa del genere:

```
    MouseInit bottoni%
    MouseSetPos 1, 1
```

Il servizio AX=4 è collegato a due altri servizi, usati di rado ma che risultano utili in determinate situazioni; si tratta del servizio AX=7 che imposta dei limiti destro e sinistro per il movimento del cursore del mouse, e del servizio AX=8 che invece imposta dei limiti in senso verticale. Poiché questi servizi sono sempre usati insieme, ho creato una procedura che li richiama in sequenza (per la precisione le procedure sono due, una per il modo testo e una per i modi grafici) e che permette quindi di restringere il movimento del mouse ad un'area rettangolare di schermo. Queste procedure sono utili quando si clicca con il mouse su una scroll bar, oppure quando si sta trascinando con il mouse un oggetto (dragging) che non può essere depositato al di fuori della finestra attiva.

```
SUB MouseTrap (riga1%, col1%, riga2%, col2%)
    '-------------------------------
    ' restringe il movimento del mouse ad un'area
    ' rettangolare (modo testo)
    '-------------------------------
    DIM reg AS regType
    reg.cx = (col1% - 1) * 8
    reg.dx = (col2% - 1) * 8
    reg.ax = 7
    Interrupt &H33, reg, reg
    reg.cx = (riga1% - 1) * 8
    reg.dx = (riga2% - 1) * 8
    reg.ax = 8
    Interrupt &H33, reg, reg
END SUB
SUB MouseTrapG (x1%, y1%, x2%, y2%)
    '-------------------------------
    ' restringe il movimento del mouse ad un'area
    ' rettangolare (modo grafico)
    '-------------------------------
    DIM reg AS regType
```

```
  reg.cx = x1%
  reg.dx = x2%
  reg.ax = 7
  Interrupt &H33, reg, reg
  reg.cx = y1%
  reg.dx = y2%
  reg.ax = 8
  Interrupt &H33, reg, reg
END SUB
```

LEGGERE LO STATO DEL MOUSE

Ovviamente, l'operazione più interessante che un programma può effettuare su un mouse e la lettura della posizione corrente del cursore e lo stato dei bottoni. Queste informazioni possono essere ottenute con un'unica chiamata al servizio AX=3, e come al solito è utile creare due procedure separate, una per il modo testo ed una per i modi grafici:

```
SUB MouseGet (riga%, col%, bottone%)
  '________________________
  ' Legge lo stato corrente del mouse (modo testo)
  '________________________
  DIM reg AS regType
  reg.ax = 3
  Interrupt &H33, reg, reg
  bottone% = reg.bx
  riga% = reg.dx \ 8 + 1
  col% = reg.cx \ 8 + 1
END SUB
SUB MouseGetG (x%, y%, bottone%)
  '________________________
  ' Legge lo stato corrente del mouse (modo grafico)
  '________________________
  DIM reg AS regType
  reg.ax = 3
  Interrupt &H33, reg, reg
  bottone% = reg.bx
  x% = reg.cx
  y% = reg.dx
END SUB
```

In entrambi i casi il valore restituito in *bottone* è codificato a bit, come segue:

bit 0 bottone sinistro premuto

bit 1 bottone destro premuto

bit 2 bottone centrale premuto

Ecco allora come scrivere un programma BASIC che sfrutta la procedura **MouseGet**:

```
MouseGet riga%, col%, bottone%
IF bottone% AND 1 THEN PRINT "Premuto tasto sinistro"
IF bottone% AND 2 THEN PRINT "Premuto tasto destro"
IF bottone% AND 4 THEN PRINT "Premuto tasto centrale"
```

Nei programmi che interagiscono con il mouse è molto frequente la necessità di controllare se un certo bottone del mouse è stato "cliccato" oppure "rilasciato"; questo controllo impone il continuo confronto tra il valore restituito da **MouseGet** con quello restituito dalla chiamata precedente, per vedere se e quali bottoni sono stati premuti o rilasciati. Il codice che esegue questo controllo non è molto complesso, grazie ad un operatore XOR che permette di determinare immediatamente quali bit in **bottone%** hanno cambiato stato, passando da zero ad uno (il bottone è stato premuto) o da uno a zero (il bottone è stato rilasciato):

```
' questo programma mostra come è possibile usare la routine
' MouseGet per determinare su un bottone è stato premuto o rilasciato
' prima chiamata a MouseGet
MouseGet riga%, col%, bottone%
' salva lo stato in una variabile
bottone2% = bottone%
...
' seconda chiamata a MouseGet
MouseGet riga%, col%, bottone%
click% = bottone% XOR bottone2%
' a questo punto la variabile click% contiene dei bit pari ad 1
' in corrispondenza dei bottoni che hanno cambiato stato, e che
' quindi sono stati cliccati o rilasciati
IF click% AND 1 THEN
  IF bottone% AND 1 THEN
    PRINT "Bottone sinistro premuto"
  ELSE
    PRINT "Bottone sinistro rilasciato"
  END IF
END IF
IF click% AND 2 THEN
  IF bottone% AND 2 THEN
    PRINT "Bottone destro premuto"
  ELSE
    PRINT "Bottone destro rilasciato"
  END IF
END IF
IF click% AND 4 THEN
  IF bottone% AND 4 THEN
    PRINT "Bottone centrale premuto"
  ELSE
    PRINT "Bottone centrale rilasciato"
  END IF
END IF
```

In alcuni casi possibile fare a meno di un programma del genere, in quanto il driver del mouse mette a disposizione due servizi che permettono di determinare quante volte un determinato bottone è stato premuto e rilasciato rispettivamente. I due servizi sono analoghi, nel senso che i valori nei registri hanno lo stesso significato, e varia solo il fatto che il primo si riferisce al click e il secondo al rilascio del bottone specificato in ingresso:

AX = 5 (click) oppure 6 (release)

BX = il bottone di cui si richiedono le informazioni (0=sinistro, 1=destro, 2=centrale)

in uscita:

AX = lo stato corrente dei bottoni

BX = il numero di click (o release) dall'ultima chiamata

CX = la posizione X dell'ultimo click (o release)

DX = la posizione Y dell'ultimo click (o release)

si noti che AX restituisce lo stato dei bottoni corrente (che è la stessa informazione restituita dal servizio 3 e quindi dalla procedura **MouseGet**), mentre CX e DX si riferiscono alla posizione del mouse al momento dell'ultima azione effettuata; inoltre, ad ogni chiamata ad uno di questi servizi viene azzerato il contatore relativo a ciascun bottone, per cui richiamando nuovamente il servizio si otterrebbe un valore nullo. Il seguente listato propone quattro procedure che possono essere utilizzate direttamente nei programmi BASIC, due per i modi testo e due per i modi grafici.

```
SUB MouseClick (bottone%, contatore%, riga%, col%)
  '___________________________
  ' Informazioni sui click del mouse (modo testo)
  ' bottone% deve essere uno dei seguenti
  '       0  = sinistro
  '       1  = destro
  '       2  = centrale
  '___________________________
  DIM reg AS regType
  reg.ax = 5
  Interrupt &H33, reg, reg
  contatore% = reg.bx
  riga% = reg.dx \ 8 + 1
  col% = reg.cx \ 8 + 1
END SUB
SUB MouseClickG (bottone%, contatore%, x%, y%)
  '___________________________
  ' Informazioni sui click del mouse (modo grafico)
  '___________________________
  DIM reg AS regType
  reg.ax = 5
  Interrupt &H33, reg, reg
  contatore% = reg.bx
  x% = reg.cx
  y% = reg.dx
END SUB
SUB MouseRelease (bottone%, contatore%, riga%, col%)
  '___________________________
  ' Informazioni sul rilascio del mouse (modo testo)
  '___________________________
  DIM reg AS regType
  reg.ax = 6
  Interrupt &H33, reg, reg
  contatore% = reg.bx
  riga% = reg.dx \ 8 + 1
  col% = reg.cx \ 8 + 1
END SUB
```

```
SUB MouseReleaseG (bottone%, contatore%, x%, y%)
  '____________________________
  ' Informazioni sul rilascio del mouse (modo grafico)
  '____________________________
  DIM reg AS regType
  reg.ax = 6
  Interrupt &H33, reg, reg
  contatore% = reg.bx
  x% = reg.cx
  y% = reg.dx
END SUB
```

Nella maggior parte dei programmi il controllo del mouse avviene con una certa frequenza - ad es. all'interno di un loop eseguito più volte per secondo - per cui è possibile dare per scontato che la posizione del mouse al momento del click o del rilascio sia la stessa di quella determinata dalla procedura **MouseGet**, per cui è possibile effettuare il controllo con due sole funzioni, che funzionano indifferentemente in modo testo e grafico (listato 4).

```
' determina se un bottone è stato premuto o rilasciato
' dall'ultima chiamata alla medesima funzione
FUNCTION MouseClicked% (bottone%)
  '____________________________
  ' restituisce -1 se il bottone è stato cliccato
  '____________________________
  DIM reg AS regType
  reg.ax = 5
  Interrupt &H33, reg, reg
  MouseClicked% = (reg.bx <> 0)
END FUNCTION
FUNCTION MouseReleased% (bottone%)
  '____________________________
  ' restituisce -1 se il bottone è stato rilasciato
  '____________________________
  DIM reg AS regType
  reg.ax = 6
  Interrupt &H33, reg, reg
  MouseReleased% = (reg.bx <> 0)
END FUNCTION
```

IL CURSORE IN MODO TESTO

Finora abbiamo dato per scontato che il cursore del mouse, in modo testo, apparisse come un blocchetto colorato in "reverse" (ottenuto invertendo il colore della casella) e trasparente, nel senso che non influisce sul carattere ASCII mostrato sotto il cursore. Il realtà l'aspetto del cursore che può essere modificato come meglio si desidera, sia nel colore che nel carattere mostrato.

Esistono in realtà due tipi di cursore mouse per il modo testo: il cursore hardware e il cursore software. I due tipi di cursore possono essere distinti con una semplice occhiata, in quanto il primo lampeggia e il secondo no. Vediamo subito come impostare il cursore hardware

```
SUB SetHardwareCursor (startLine%, endLine%)
  '________________________
  ' Imposta il cursore hardware per il mouse
  '________________________
  DIM reg AS RegTypeX
  reg.ax = &HA        ' servizio 0A hex
  reg.bx = 1         ' 1 = cursore hardware
  reg.cx = startLine%    ' dimensioni del cursore in
  reg.dx = endLine%     ' CX e DX
  InterruptX &H33, reg, reg  ' richiama il driver del mouse
END SUB
```

Quando si definisce il cursore hardware occorre indicare la linea iniziale e finale di scansione, in modo molto simile alla definizione del cursore per i caratteri con l'istruzione LOCATE; anche nel caso del mouse i valori massimi per i parametri **startLine%** e **endLine%** dipendono dalla particolare scheda video utilizzata.

In realtà, la maggior parte dei programmi per Ms-Dos sfrutta invece il cursore software, che permette una maggiore versatilità. Ecco una routine che attiva il cursore software e ne definisce le caratteristiche:

```
SUB SetSoftwareCursor (andMask%, xorMask%)
  '________________________
' Attiva il cursore software per il mouse
  '________________________
  DIM reg AS RegTypeX
  reg.ax = &HA        ' servizio 0A hex
  reg.bx = 0        ' 0 = cursore software
  reg.cx = andMask%     ' maschera AND in CX
  reg.dx = xorMask%     ' maschera XOR in DX
  InterruptX &H33, reg, reg  ' richiama il driver del mouse
END SUB
```

Per capire il significato dei due argomenti **andMask%** e **xorMask%** occorre ricordare che nei modi testi le informazioni conservate in memoria video sono organizzate a coppie di byte

```
byte pari  byte dispari
(carattere)  (colore)
AAAAAAAA  FSSSTTTT
```

dove AAAAAAAA sono i vari bit che compongono il codice ASCII del carattere mostrato in ciascuna posizione, F è il bit di lampeggio, SSS indica il codice del colore di sfondo (nell'intervallo 0-7) e TTTT il codice del colore del testo (nell'intervallo 0-15). Il driver che gestisce il mouse mostra il cursore software eseguendo una doppia operazione sulla coppia di byte corrispondenti alla posizione in cui si trova il mouse stesso, corrispondente alla seguente operazione BASIC:

```
nuovoValore% = (valoreCorrente% AND andMask%) XOR xorMask%
```

È evidente, quindi, che usando opportuni valori per le due maschere di bit è possibile ottenere qualsiasi combinazione di carattere e di colore. Cercherò di chiarire il concetto con qualche esempio:

```
SetSoftwareCursor &HFFFF, &H7F00
```

se il valore di **andMask%** è pari a FFFFh, la prima istruzione AND non modifica il valore corrente, mentre l'operazione di XOR con 7F00 lascia inalterato il codice del carattere ma complementa sia il colore del testo che il colore dello sfondo. Ad esempio, il colore di sfondo nero (codice = 0) diventerà bianco (codice = 7), mentre un testo di colore blu (codice = 1) sarà mostrato in giallo intenso (codice = 14), e viceversa.

```
SetSoftwareCursor &HFFFF, &H8000
```

in questo caso sia il codice del carattere che il colore del testo e dello sfondo non solo alterati; l'unico tratto distintivo di questo tipo di cursore sarà dato dal lampeggio del carattere sotto il cursore del mouse.

```
SetSoftwareCursor &HF000, &H0F2A
```

in questo caso il colore di sfondo è inalterato, ma il cursore è mostrato come un asterisco (codice esadecimale 2A) in bianco brillante (codice 0F).

IL CURSORE IN MODO GRAFICO

Nei modi grafici il cursore di default ha la forma di una freccia con la punta in alto a sinistra, ma anche in questo caso è possibile modificare l'aspetto come meglio si crede, usando il servizio 9 del driver. Prima di richiamare questa funzione dobbiamo però creare due maschere di bit. Ambedue le maschere sono di 16 x 16 bit, dove ciascun bit corrisponde ad un pixel dello schermo; poiché una riga di 16 pixel può essere rappresentata da una word (due byte), possiamo anche dire che ciascuna maschera sarà lunga 16 word.

Ogni volta che il mouse è spostato in una nuova posizione dello schermo il driver effettua una operazione di AND tra la prima maschera e i pixel in quella particolare posizione dello schermo, immediatamente seguita da una operazione di XOR con le sedici word che compongono la seconda maschera. Si tratta quindi di un procedimento simile a quello visto per il cursore in modo testo, con la importante differenza che in questo caso trattiamo con i singoli pixel e non con gli attributi di colore o il codice ASCII del carattere.

Tenuto conto che per ciascun pixel delle due maschere sono possibili quattro combinazioni, vediamo che:

se un bit è 0 nella maschera AND e 0 nella maschera XOR, il corrispondente pixel sullo schermo sarà di colore nero

se un bit è 0 nella maschera AND e 1 nella maschera XOR, il corrispondente pixel sullo schermo sarà di colore bianco

se un bit è 1 nella maschera AND e 0 nella maschera XOR il corrispondente pixel sullo schermo non è influenzato dal mouse (trasparente)

se un bit è 1 nella maschera AND e 1 nella maschera XOR il colore del corrispondente pixel sullo schermo viene invertito (da bianco a nero, e viceversa)

Facciamo un esempio, usando delle maschere di bit 0 e 1; ecco ad esempio un cursore a forma di croce bianca contornata da una riga di pixel neri:

```
1111111111111111   0000000000000000
1111111111111111   0000000000000000
1111110001111111   0000000000000000
1111110001111111   0000000100000000
1111110001111111   0000000100000000
1111110001111111   0000000100000000
1000000000000011   0000000100000000
1000000000000011   0011111111111000
1000000000000011   0000000100000000
1111110001111111   0000000100000000
1111110001111111   0000000100000000
1111110001111111   0000000100000000
1111110001111111   0000000000000000
1111111111111111   0000000000000000
1111111111111111   0000000000000000
1111111111111111   0000000000000000
```

Le due maschere di bit devono essere memorizzate in un'area di memoria una di seguito all'altra, per cui occorre un buffer lungo almeno 64 byte; in BASIC la soluzione più comoda è usare un array di INTEGER di 32 elementi. Infine occorre fornire al driver del mouse la posizione del cosiddetto *hot spot*, ossia del pixel che corrisponde alla reale posizione del mouse, come può essere il caso della punta della freccia nella forma di default oppure il centro della croce nell'esempio appena visto. Ecco una routine che modifica la forma del cursore grafico:

```
SUB SetGraphicCursor (xOffset%, yOffset%, mask%())
  '____________________________
  ' Imposta la forma del cursore grafico
  '____________________________
  DIM reg AS RegTypeX
  reg.ax = 9        ' servizio 9
  reg.bx = xOffset%    ' offset orizzontale in BX
  reg.cx = yOffset%    ' offset verticale in CX
  reg.es = VARSEG(mask%(0))  ' indirizzo delle maschere
  reg.dx = VARPTR(mask%(0))  ' in ES:DX
  InterruptX &H33, reg, reg  ' richiama il driver del mouse
END SUB
```

Si noti che il valore di **xOffset** e **yOffset** può variare tra -16 e 16; se uno dei due assume valore negativo l'hot spot risulta essere esterno alla maschera di bit; inoltre in alcuni modi grafici l'offset orizzontale dovrebbe essere un numero pari.

Ed ecco come usare la routine appena definita per creare il nostro cursore a forma di croce:

```
DIM mask%(31)
FOR i% = 0 TO 31
```

```
  READ mask%(i)
NEXT
SetGraphicCursor 7, 7, mask%()
' questa è la maschera AND
DATA &HFFFF, &HFFFF, &HFA7F, &HFA7F, &HFA7F, &HFA7F, &H8002, &H8002
DATA &H8002, &HFA7F, &HFA7F, &HFA7F, &HFA7F, &HFFFF, &HFFFF, &HFFFF
' questa è la maschera XOR
DATA &H0000, &H0000, &H0000, &H0100, &H0100, &H0100, &H0100, &H3FF8
DATA &H0100, &H0100, &H0100, &H0100, &H0000, &H0000, &H0000, &H0000
```

dove per comodità i bit sono stati convertiti in valori esadecimali. Per aiutarvi ad inserire direttamente i valori negli elementi dell'array **mask%()** ho scritto una piccola routine di conversione:

```
FUNCTION BinToDec% (value$)
  '________________________
  ' Conversione da binario a decimale
  '________________________
  DIM result&, i%
  FOR i% = 1 TO LEN(value$)
    result& = result& * 2 + VAL(MID$(value$, i%, 1))
  NEXT
  ' converti in un integer a 16 bit
  BinToDec% = CVI(MKL$(result&))
END FUNCTION
```

Ecco un frammento di programma che sfrutta la funzione **BinToHex$** per definire un cursore a forma di mirino (un cerchio con una croce in mezzo):

```
DIM mask%(31)
FOR i% = 0 TO 31
  READ temp$
  mask%(i) = BinToDec%(temp$)
NEXT
SetGraphicCursor 7, 7, mask%()
DATA 1111111111111111
DATA 1111111111111111
DATA 1111110001111111
DATA 1111001010011111
DATA 1100111011100111
DATA 0011111011111001
DATA 0000000000000001
DATA 0011111011111001
DATA 1100111011100111
DATA 1111001010011111
DATA 1111110001111111
DATA 1111111111111111
DATA 1111111111111111
DATA 1111111111111111
DATA 1111111111111111
DATA 1111111111111111
DATA 0000000000000000
DATA 0000000000000000
DATA 0000001110000000
DATA 0000110101100000
DATA 0011000100011000
DATA 1100000100000110
DATA 1111111111111110
DATA 1100000100000110
DATA 0011000100011000
DATA 0000110101100000
```

```
DATA 0000001110000000
DATA 0000000000000000
DATA 0000000000000000
DATA 0000000000000000
DATA 0000000000000000
DATA 0000000000000000
```

LA SENSIBILITÀ DEL MOUSE

Alcuni servizi del driver permettono di calibrare la sensibilità del mouse, ossia il rapporto tra il movimento "reale" sul tavolo o sul mouse pad e la distanza in pixel realmente percorsa sullo schermo. Per misurare il movimento reale del mouse qualche spiritosone ha inventato una nuova unità di misura, il *mickey*, equivalente a circa 1/200 di pollice (cioè 0.127 millimetri). Il servizio 0Fh permette di impostare il rapporto tra il movimento in mickey e lo spostamento risultante in pixel; gli argomenti indicano il numero di mickey necessari per ottenere uno spostamento di 8 pixel nelle due direzioni:

```
SUB SetMouseSensibility (horizontal%, vertical%)
  '___________________________
  ' Imposta la sensibilità del mouse
  '___________________________
  DIM reg AS RegType
  reg.ax = &HF        ' servizio 0F del driver
  reg.cx = horizontal%    ' sensibilità oriz. in CX
  reg.dx = vertical%    ' sensibilità vert. in DX
  Interrupt &H33, reg, reg
END SUB
```

poiché 8 pixel corrispondono ad un carattere nei modi testo, possiamo anche considerare i due argomenti come lo spostamento minimo per spostare il cursore del mouse di un carattere. Entrambi gli argomenti possono variare tra 1 e 32767; i valori di default sono 8 mickey per gli spostamenti in direzione orizzontale e 16 mickey per gli spostamenti in direzione verticale.

Esistono poi i cosiddetti mouse "balistici", che permettono una maggiore manovrabilità usando l'artificio di raddoppiare automaticamente il rapporto pixel/mickey quando il mouse è spostato sul tavolo più rapidamente di una certa "velocità di soglia". Nei mouse di questo tipo la velocità di soglia (*threshold speed*) impostata per default è di 64 mickey per secondo, ma può essere modificata con una chiamata al servizio 13h:

```
SUB SetThresholdSpeed (speed%)
  '_______________________________
  ' Imposta la velocità di soglia per i mouse balistici
  '_______________________________
  DIM reg AS RegType
  reg.ax = &H13       ' servizio 13 del driver
  reg.dx = speed%     ' velocità in DX
  Interrupt &H33, reg, reg
END SUB
```

La doppia velocità può essere del tutto disabilitata richiamando la procedura **SetThresholdSpeed** con un valore molto alto come argomento (ad esempio 30000).

Il driver fornisce anche un servizio per leggere i valori correnti dei parametri appena visti, in modo che un programma sia in grado di alterare i valori correnti e di ripristinarli prima di terminare e cedere il controllo al sistema operativo.

```
SUB GetMouseSensibility (horizontal%, vertical%, threshold%)
  '______________________________
  ' Legge i valori di sensibilità del mouse
  '______________________________
  DIM reg AS RegType
  reg.ax = &H1B       ' servizio 1B del driver
  Interrupt &H33, reg, reg
  horizontal% = reg.bx     ' restituisce i valori in
  vertical% = reg.cx     ' BX, CX e DX
  threshold% = reg.dx
END SUB
```

SALVARE E RIPRISTINARE LO STATO DEL DRIVER

In alcuni casi può essere necessario salvare lo stato corrente del driver - che comprende le informazioni riguardanti la posizione e la forma del cursore, la sensibilità, la velocità di soglia, ecc. - per poi ripristinarle in seguito. L'esempio più evidente ce lo offre proprio l'ambiente di sviluppo del BASIC, che usa il mouse per le operazioni di editing ma che permette ai nostri programmi BASIC di utilizzare anch'essi questa periferica: se provate ad eseguire un vostro applicativo che fa uso del mouse all'interno dell'interprete ed impostate un breakpoint, vedrete che all'apparire dell'editor il mouse tornerà nella posizione in cui lo avevamo lasciato prima di far partire il nostro programma; se a questo punto premiamo F5 per fare continuare l'esecuzione, tornerà ad essere mostrato il cursore mouse relativo al nostro applicativo.

Il driver mette a disposizione tre servizi per salvare e ripristinare lo stato del driver; il primo servizio restituisce la dimensione delle informazioni memorizzate nel driver:

```
FUNCTION MouseBufferSize%
  '______________________________
  ' restituisce la dimensione del buffer interno
  '______________________________
  DIM reg AS RegType
  reg.ax = &H15       ' servizio 15 del driver
  Interrupt &H33, reg, reg
  MouseBufferSize% = reg.bx  ' restituisce la dimensione in BX
END SUB
```

La funzione appena vista permette di dimensionare un buffer di dimensioni adeguate, per poi leggere lo stato corrente con una chiamata al servizio 16h del driver; come al solito useremo un vettore di integer:

```
SUB SaveMouseState (buffer%())
  '______________________________
  ' Legge lo stato interno del driver
  '______________________________
  DIM reg AS RegTypeX
  reg.ax = &H16        ' servizio 16 del driver
  reg.es = VARSEG(buffer%(0))' richiede l'indirizzo del
  reg.dx = VARPTR(buffer%(0))' buffer in ES:DX
  InterruptX &H33, reg, reg
END SUB
```

Scriveremo quindi qualcosa del genere:

```
DIM buffer%(MouseBufferSize% \ 2)
SaveMouseState buffer%()
```

Per ripristinare lo stato salvato in precedente abbiamo bisogno di un'altra routine, molto simile alla precedente:

```
SUB RestoreMouseState (buffer%())
  '______________________________
  ' Ripristina lo stato interno del driver
  '______________________________
  DIM reg AS RegTypeX
  reg.ax = &H17        ' servizio 17 del driver
  reg.es = VARSEG(buffer%(0))' richiede l'indirizzo del
  reg.dx = VARPTR(buffer%(0))' buffer in ES:DX
  InterruptX &H33, reg, reg
END SUB
```

La possibilità di salvare e ripristinare lo stato del driver è molto utile quando si lanciano programmi esterni per mezzo di una istruzione SHELL.

INFORMAZIONI SUL DRIVER

Il driver fornisce anche un servizio a scopo diagnostico, che può essere usato per ricavare informazioni sul tipo e sulla versione del driver:

```
SUB GetMouseInfo (version%, mouseType%, irqNumber%)
  '______________________________
  ' Restituisce informazioni sul driver
  '______________________________
  DIM reg AS RegType
  reg.ax = &H24        ' servizio 24 del driver
  Interrupt &H33, reg, reg
  version% = (reg.bx \ 256) * 100 + (reg.bx AND 255)
  mouseType% = reg.cx \ 256
  irqNumber% = (reg.cx AND 255)
END SUB
```

Il numero di versione è restituito nei due registri BH e BL, e la routine precedente combina i due valori per ottenere un numero intero a tre cifre (ad

es. 620 per la versione 6.2); il parametro **mouseType%** restituisce un intero secondo lo schema seguente:

1	bus mouse
2	mouse serial
3	mouse InPort
4	mouse PS/2
5	mouse HP

Il valore restituito da **irqNumber%** corrisponde all'interrupt IRQ usato dal mouse ed è compreso tra uno dei valori seguenti: 2, 3, 4, 5 o 7; il mouse PS/2 non usa IRQ e in tal caso è restituito un valore nullo per questo argomento.

UN ESEMPIO PRATICO

E' arrivato il momento di vedere qualche applicazione pratica di quanto detto fino a questo punto. Iniziamo con qualcosa di semplice, un piccolo programma che controlla il mouse e mostra continuamente sulla riga in basso le coordinate correnti del cursore; pur nella sua banalità, questo piccolo problema ci permette di capire abbastanza velocemente quali sono i problemi che si incontrano nella programmazione del mouse, tanto che invece di fornire la soluzione completa procederò per versioni successive. Il programma è un semplice dimostrativo che mostra continuamente le coordinare del cursore sul rigo in basso:

```
' Versione n.1
MouseShow
LOCATE , , 1
DO
  MouseGet bottone%, riga%, col%
  LOCATE 25, 1
  PRINT "Riga "; riga%; "  Colonna "; col%; "  ";
LOOP UNTIL INKEY$ = CHR$(27)
MouseHide
```

La prima versione è la più immediata, ma presenta almeno tre problemi: (a) il ciclo non è molto efficiente, in quanto mostra continuamente il medesimo messaggio anche se il mouse non è stato mosso; (b) il cursore del mouse scompare quando si sovrappone al messaggio nell'ultima riga; (c) essendo attivo il cursore del modo testo (vedi la prima LOCATE) assisteremo ad un continuo sfarfallio, dovuto al continuo spostamento del cursore stesso. Ovviamente in questo caso l'ultimo problema può essere evitato disattivando il cursore hardware, ma nella maggior parte delle applicazioni questa soluzione non è praticabile perché entrambi i cursori devono essere ben visibili.

Ambedue i problemi sono risolti nella seconda versione, che fa uso di due variabili per memorizzare le coordinate correnti del mouse e che visualizza il messaggio solo se queste cambiano, disabilitando e riabilitando il cursore del mouse per evitare il problema (b), mentre gli altri due svantaggi sono automaticamente evitati riducendo il numero di volte in cui il messaggio è mostrato:

```
' versione n.2
MouseShow
LOCATE , , 1
DO
  riga2% = riga%: col2% = col%
  MouseGet bottone%, riga%, col%
  IF riga% <> riga2% OR col% <> col2% THEN
    MouseHide
    LOCATE 25, 1, 0
    PRINT "Riga "; riga%; "  Colonna "; col%; "  ";
    MouseShow
  END IF
LOOP UNTIL INKEY$ = CHR$(27)
MouseHide
```

Complichiamo ulteriormente il problema, considerando il caso in cui il programma debba gestire un cursore hardware indipendente; è evidente che occorre ogni volta salvare e ripristinare le coordinate del cursore prima di stampare il messaggio sulla riga in basso; il listato della terza versione del nostro programma offre inoltre la possibilità di spostare il cursore hardware premendo il bottone sinistro; notate che non viene controllato il semplice click, in quanto il cursore deve essere mosso anche quando il mouse è "trascinato" tenendo premuto il bottone.

```
' versione n.3
MouseShow
LOCATE , , 1
DO
  riga2% = riga%: col2% = col%
  MouseGet bottone%, riga%, col%
  IF bottone% AND 1 THEN LOCATE riga%, col%
  IF riga% <> riga2% OR col% <> col2% THEN
    rigaCurs% = CSRLIN: colCurs% = POS(0)
    MouseHide
    LOCATE 25, 1, 0
    PRINT "Riga "; riga%; "  Colonna "; col%; "  ";
    LOCATE rigaCurs%, colCurs%, 1
    MouseShow
  END IF
LOOP UNTIL INKEY$ = CHR$(27)
MouseHide
```

IL DRAG-AND-DROP

Ormai i programmi Windows ci hanno abituato a questa caratteristica del mouse, che consiste nel "afferrare" un oggetto sullo schermo (ad es. una icona)

e trascinarlo in un altro punto dello schermo; quando si "lascia cadere" l'oggetto rilasciando il bottone del mouse il programma risponde con una azione. L'esempio classico è il trascinamento del nome di un file sull'icona del cestino della carta straccia per ottenerne la cancellazione. Anche in Dos è possibile ottenere qualcosa di simile, a condizione però di accollarsi tutto l'onere delle varie operazioni necessarie, che invece sotto Windows è a carico del sistema.

Il nostro esempio di drag & drop consisterà in un programma che permette di selezionare dei nomi da una lista per costruire un elenco di squadre, e potrebbe essere il punto di partenza per un programma più complesso. Il drag & drop permetterà all'operatore di spostare i nomi sullo schermo nel modo più naturale possibile, agganciandoli con il mouse e depositandoli dove più ci aggrada; come potete vedere che segue, il ciclo che provvede alla gestione del mouse conta appena una ottantina di istruzioni (commenti a parte), forse addirittura meno di quelle necessarie per ottenere le stesse prestazioni usando esclusivamente la tastiera.

Anche nella sua brevità il programma è sufficientemente completo, in quanto gestisce in modo appropriato tutte le situazioni e i problemi che possono verificarsi lavorando con il drag & drop:

a) attivazione del drag, che avviene quando il mouse è cliccato su un nome tra quelli nella lista - il programma gestisce questa situazione conservando in due vettori la riga e la colonna corrispondenti alla posizione attuale del nome sullo schermo (vettori **nomiR** e **nomiC**); per un migliore feedback all'utente, il nome selezionato viene visualizzato in reverse

b) spostamento del nome selezionato - è relativamente semplice fare in modo che il nome selezionato "segua" il mouse se l'operatore continua a mantenere premuto il bottone sinistro; il problema della ricostruzione del contenuto dello schermo è risolto salvando con PCOPY la situazione corrente all'inizio della fase (a) e ripristinandola prima di ridisegnare il nome selezionato nella nuova posizione

c) l'operazione di drop - in questo caso è necessario controllare che il drop avvenga in un'area libera dello schermo, in quanto non ha senso lasciar cadere un nome su un'altro nome; le operazioni di drop non consentite sono segnalate mediante un bip, che continua fino a quando il mouse non è spostato in un'area libera dello schermo

d) oltre alle operazioni di drag & drop, il programma mostra come è possibile controllare il click del mouse su eventuali bottoni presenti sullo schermo

Come vedete, ce n'è per tutti i gusti e necessità; è facile vedere, ad esempio, che le operazioni descritte al punto (c) possono essere modificate per controllare se il drop avviene in una particolare area dello schermo, ad esempio una icona che rappresenta la stampante o il cestino dei rifiuti, in modo da attivare particolari comandi.

in Visual BASIC *per* Dos *le operazioni di drag & drop, come del resto tutte le operazioni con il mouse sono gestite direttamente dal linguaggio, che in un certo senso funge da "cuscinetto" tra il mouse e la applicazione e simula alcuni dei servizi che offre* Windows *ai propri programmi.* Per *questo motivo, tutto quanto detto finora sul mouse in generale e sul drag & drop in particolare non vale per i form del* Visual BASIC *per* Dos.

```
' DEMODRAG.BAS
' Programma dimostrativo del drag & drop in BASIC
' in questo listato non sono riportate le varie
' procedure di supporto per il mouse
'
' NOTA: per ragioni tipografiche alcune righe sono
'       state spezzate, ma dovrebbero essere inserite
'       come un'unica linea di programma
'$INCLUDE: 'qbx.bi'
CONST NOMIMAX = 20
DEFINT A-Z
DIM nomi$(NOMIMAX), nomiR(NOMIMAX), nomiC(NOMIMAX)
' disegna le stringhe fisse sullo schermo
CLS
LOCATE , , 0
LOCATE 1, 8: PRINT "GIOCATORI"
LOCATE 1, 40: PRINT "SQUADRE"
LOCATE 2, 1: PRINT STRING$(80, 196)
FOR i = 1 TO 10
  LOCATE i + 3, 35: PRINT LTRIM$(STR$(i)); ")";
NEXT
LOCATE 24, 1: PRINT STRING$(80, 196);
LOCATE 20, 60: PRINT ""
LOCATE 21, 60: PRINT "    OK    "
LOCATE 22, 60: PRINT ""
' leggi il vettore nei nomi e mostrali sullo schermo
' inizializza i vettori
FOR i = 1 TO NOMIMAX
  READ nomi$(i)
  nomiR(i) = i + 3
  nomiC(i) = 8
  LOCATE nomiR(i), nomiC(i)
  PRINT nomi$(i);
NEXT
DATA "Adriana", "Berto", "Claudia", "Daniele", "Enrica", "Francesco"
DATA "Giovanna","Helen", "Ilaria", "Lello", "Maria", "Nicola"
DATA "Olga", "Pietro", "Renato", "Silvia", "Teresa", "Umberto"
DATA "Vincenzo", "Zoe"
MouseInit bottoni
IF bottoni = 0 THEN
  PRINT "Questo programma richiede il mouse"
```

```
  END
END IF
MouseShow
DO
  riga2 = riga: col2 = col: clicked2 = clicked
  MouseGet bottone, riga, col
  clicked = bottone AND 1
  IF clicked <> 0 AND clicked2 = 0 AND selez = 0 THEN
    ' il mouse è stato cliccato
    ' esci se se si tratta del bottone OK
    IF riga >= 20 AND riga <= 22 AND col >= 60 AND col <= 71 THEN
      EXIT DO
    END IF
    ' controlla se il cursore è su un nome di persona
    FOR i = 1 TO NOMIMAX
      ' (la seguente riga è spezzata per ragioni tipografiche)
      IF riga = nomiR(i) AND col >= nomiC(i) AND _
          col < nomiC(i) + LEN(nomi$(i)) THEN
        selez = i: EXIT FOR
      END IF
    NEXT
    ' se è stato selezionato un nome, cancellalo dallo schermo
    ' salva lo schermo sottostante, e mostra il nome in reverse
    IF selez THEN
      MouseHide
      LOCATE nomiR(selez), nomiC(selez)
      PRINT SPACE$(LEN(nomi$(selez)));
      PCOPY 0, 1
      COLOR 0, 7
      LOCATE nomiR(selez), nomiC(selez)
      PRINT nomi$(selez);
      COLOR 7, 0
      MouseShow
    ELSE
      BEEP
    END IF
  ELSEIF clicked = 0 AND selez <> 0 THEN
    ' operazione di "drop"
    ' controlla che il nome selezionato non si sovrapponga
    ' ad un altro nome
    overlap = 0
    FOR i = 1 TO NOMIMAX
      IF i <> selez AND nomiR(i) = nomiR(selez) THEN
        ' (la seguente riga è spezzata per ragioni tipografiche)
        IF nomiC(i) < nomiC(selez) + LEN(nomi$(selez)) AND _
            nomiC(i) + LEN(nomi$(i)) > nomiC(selez) THEN
          overlap = i
          EXIT FOR
        END IF
      END IF
    NEXT
    ' esegue il drop solo se non vi è sovrapposzione
    IF overlap = 0 THEN
      MouseHide
      PCOPY 1, 0
      LOCATE nomiR(selez), nomiC(selez)
      PRINT nomi$(selez);
      selez = 0
      MouseShow
    ELSE
```

```
      BEEP
    END IF
  END IF
  IF selez > 0 AND (riga <> riga2 OR col <> col2) THEN
    ' il mouse è stato mosso mentre un elemento è selezionato
    ' ripristina lo schermo precedente
    MouseHide
    PCOPY 1, 0
    ' aggiorna la posizione del nome sullo schermo
    ' controllando che non esca dallo schermo
    nomiR(selez) = riga
    col2 = nomiC(selez) + col - col2
    IF col2 > 0 AND col2 + LEN(nomi$(selez)) - 1 <= 80 THEN
      nomiC(selez) = col2
    END IF
    ' ridisegna il nome nella nuova posizione
    COLOR 0, 7
    LOCATE nomiR(selez), nomiC(selez)
    PRINT nomi$(selez);
    COLOR 7, 0
    MouseShow
  END IF
LOOP
MouseHide
```

6

LA MEMORIA ESPANSA

Poco più di una decina di anni fa i personal computer più diffusi erano i sistemi 8086 con 128K o 256K di memoria convenzionale; ora ci sembra normale disporre di un 80386 o superiore con almeno 4MB di memoria. Purtroppo il BASIC non ha tenuto il passo con le innovazioni della tecnica, e anche se è possibile creare programmi con overlay o eseguire CHAIN tra moduli dello stesso programma, non è sbagliato dire che le applicazioni scritte in questo linguaggio sono ancora legate al famoso limite dei 640K.

La situazione è ancora più seccante poiché il cliente sa bene di avere alcuni megabyte di memoria da qualche parte (perché li ha pagati...), e siamo costretti ogni volta a intraprendere lunghe spiegazioni tecniche sulla distinzione tra memoria convenzionale, espansa ed estesa, che sicuramente non risolvono il problema.

In realtà è possibile fare qualche cosa senza cambiare linguaggio di programmazione: se il programma deve lavorare con una grande quantità di dati possiamo ad esempio fare in modo di memorizzare questi dati in memoria espansa. Il codice continuerà ad essere eseguito esclusivamente in memoria convenzionale, ma almeno riusciremo a sfruttare quei famosi megabyte addizionali.

LO STANDARD EMS

In linea di principio la memoria espansa può essere disponibile su qualunque computer, incluso i sistemi 8086 e 80286; infatti, quando nella primavera 1985 fu reso pubblico lo standard EMS (detto anche standard LIM, da *Microsoft-Intel-Lotus* - le aziende che lo hanno creato, anche se in realtà Microsoft aderì solo in un secondo momento) il microprocessore 80386 non era stato ancora progettato, e anche i computer della classe AT non erano molto diffusi. Le prime specifiche 3.0 di tale standard prevedevano un sotto-sistema hardware/ software, la cui parte fisica era costituita da una scheda contenente la memoria (che non era quindi installata direttamente sulla scheda madre) e che si interfacciava alla CPU attraverso il bus di sistema. Lo standard fu migliorato nell'autunno dello stesso anno con la versione 3.2 di EMS, che aggiungeva il supporto per i sistemi multitasking e per lo stesso Windows 2.x, che fu riscritto per approfittare delle nuove possibilità offerte da questo tipo di memoria. Con la versione 4.0, rilasciata nell'estate del 1987, lo standard fu completato, e questa è la versione correntemente in uso e che da allora non è stata più modificata. Nel frattempo, la Intel aveva prodotto i primi 80386, ed alcune software house - prima tra tutte la *Qualitas* con il suo 386MAX, seguita a ruota dalla *Quaterdeck* con il noto QEMM - si resero conto che il nuovo microprocessore era in grado si simulare la memoria espansa *interamente via software*! In altre parole, con un 80386 o un processore più evoluto è possibile disporre di memoria espansa senza dover acquistare una scheda (costosa), ma semplicemente montando i chip di memoria sulla scheda madre. Questo contribuì enormemente a diffondere lo standard EMS, al punto che oggi praticamente tutti i programmi applicativi più famosi sono in grado di sfruttare tutta la memoria espansa.

Eppure non sono molti i programmatori che sfruttano regolarmente la memoria EMS nei propri programmi, soprattutto perché pochi linguaggi di programmazione permettono di accedere direttamente ad essa: il Clipper, ad esempio, può usare la memoria EMS come buffer per i file e gli indici, ma non permette di memorizzare array in EMS. Per quanto riguarda il BASIC, l'ambiente di sviluppo del BASIC PDS e del Visual BASIC per Dos sfrutta la memoria espansa per conservare le routine del programma sorgente comprese tra 1K e 16K, ed opzionalmente gli array non maggiori di 16K (usando lo switch /Es alla partenza dell'interprete), ma queste potenzialità non sono "ereditate" dai programmi applicativi scritti in questo linguaggio.

Per poter sfruttare la memoria espansa occorre prima di tutto installare un apposito driver software per poterla gestire. Il driver di questo tipo sicuramente più diffuso è il 386EMM.EXE, poiché a partire da Dos 5 esso è fornito gratuitamente con il sistema operativo (nella versione 4.0 era presente un

programma analogo, chiamato XMA2EMS.SYS). Come ogni device driver, esso deve essere specificato in una direttiva DEVICE nel file di configurazione CONFIG.SYS:

```
DEVICE = C:\DOS\EMM386.EXE  512 RAM
```

dove 512 è il numero di Kbytes di memoria estesa da convertire in memoria espansa, e RAM indica di attivare anche la memoria superiore (gli *Upper Memory Block*, o UMB). La maggior parte dei device driver più sofisticati, come i 386MAX e QEMM già citati - e lo stesso 386EMM nella versione distribuita con il nuovo MsDos 6.x - non richiedono di specificare una particolare quantità di memoria da utilizzare come espansa, in quanto si adattano automaticamente alle richieste dei programmi applicativi, ripartendo la memoria esistente nel modo migliore possibile.

Molti non sanno che per testare l'esistenza di memoria espansa non è necessario scrivere una routine in Assembly o in qualche altro linguaggio di programmazione, in quanto è sufficiente un breve programma batch:

```
IF EXIST EMMXXXX0 ECHO Memoria espansa installata
```

Continuando a leggere sarà chiaro come e perché può funzionare questa l'istruzione.

I DETTAGLI TECNICI

La memoria EMS si basa sul cosiddetto *bank switching* (alternanza tra i banchi di memoria), in cui il programma non può accedere direttamente e contemporaneamente a tutta la memoria espansa presente sul sistema, ma la lettura/scrittura dei dati avviene in una particolare "finestra" di indirizzi, situata nel primo megabyte di memoria, quasi sempre in memoria superiore tra 640K e 1M. Il meccanismo adottato nelle versioni 3.0 e 3.2 di EMS è leggermente più semplice, per cui comincerò da quest'ultimo.

Tutta la memoria espansa disponibile è suddivisa in pagine "logiche" di 16K ciascuna, numerate da 0 a (N-1), dove N è il numero totale di pagine disponibili; la "finestra" in memoria convenzionale, detta anche *frame buffer*, è larga 64K ed è quindi formata da quattro pagine "fisiche", numerate da 0 a 3. Il driver che gestisce la memoria espansa risponde alle richieste dei programmi applicativi facendo apparire una particolare pagina logica agli indirizzi di memoria di una pagina fisica, che essendo situata nel primo megabyte dello spazio di indirizzamento può essere letta e scritta dal processore senza entrare in modo protetto. Questo processo, detto *mapping*, si ripete numerose volte durante l'esecuzione di un programma, ogni volta che un dato presente in memoria EMS deve essere letto o modificato; il mapping non comporta un

vero e proprio trasferimento delle pagine di memoria, ma solo la modifica del valore di uno o più registri hardware, il cui effetto è quello di fare apparire nella finestra del frame buffer una pagina fisica piuttosto che un altra.

Nello standard 4.0 il meccanismo è praticamente identico, con la differenza che le pagine possono avere dimensioni differenti da 16K, e possono essere fatte "apparire" in qualunque zona della memoria. Inoltre, in EMS 4.0 sono disponibili fino a 32M di memoria (contro i 16M della versione 3.2) e vi sono alcune altre possibilità addizionali - tra cui quella di poter proteggere alcune pagine logiche dal reset hardware - anche se queste possibilità sono importanti solo per coloro che intendono scrivere software di sistema, device driver o sistemi operativi.

Dopo questi chiarimenti, si può rispondere ad una domanda inevitabile: poiché esistono più versioni del driver, conviene scrivere un programma utilizzando le specifiche della più recente 4.0, magari sacrificando la compatibilità con le versioni precedenti, oppure è preferibile attenersi alle caratteristiche della versione 3.0 e 3.2 ? La risposta dipende a seconda dei casi particolari, ma si tenga presente che è davvero difficile trovare in circolazione gestori di memoria che non adottano pienamente lo standard EMS 4.0, e quasi sempre si tratta di vecchie schede di memoria espansa. Possiamo quindi concludere che se il programma applicativo non ha particolari esigenze, è preferibile fare a meno dei servizi del driver 4.0, guadagnandoci in compatibilità con tutti i sistemi dotati di memoria EMS, ma se risulta vantaggioso sfruttare le caratteristiche dello standard 4.0 non si abbia troppo timore, in quanto saranno pochi i sistemi incompatibili con il software. Nella trattazione che segue indicherò sempre se un particolare servizio è disponibile sono nelle versioni più recenti del driver.

Un punto fondamentale, che dovrebbe essere ben chiaro a chi intende utilizzare la memoria EMS nei propri programmi, è che il sistema operativo è completamente ignaro del fatto che un programma sta usando o meno la memoria espansa. Questa considerazione ha una conseguenza molto importante: il sistema operativo non è in grado di intervenire per forzare il rilascio della memoria EMS utilizzata da un applicativo quando l'applicativo stesso termina. In altre parole, se il programma termina senza deallocare esplicitamente la memoria espansa utilizzata, questa memoria non sarà rilasciata e la quantità di memoria EMS libera diminuirà di conseguenza. La mancata deallocazione della memoria espansa può derivare da un errore del programmatore, oppure da una terminazione inaspettata del programma a causa di un errore o di un bug. In entrambi i casi l'effetto è il medesimo: dopo uno o più lanci consecutivi del programma la memoria espansa libera sarà ridotta a zero, e di conseguenza il programma funzionerà male o non funzionerà affatto. Un consiglio: se per qualche motivo il programma che state testando termina

senza rilasciare la memoria EMS, resettate il sistema per essere certi ogni volta di ricominciare con tutta la memoria disponibile.

LA MEMORIA EMS NEI PROGRAMMI BASIC

Se si intende utilizzare la memoria espansa nei programmi BASIC interpretati in BASIC PDS o Visual BASIC per Dos è indispensabile lanciare l'ambiente di sviluppo con l'opzione /Es:

```
QBX /Es      oppure      VBDOS /Es
```

in questo modo si segnala al BASIC che il programma applicativo sfrutta la memoria espansa, e si eviteranno conflitti che sicuramente causerebbero un crash del sistema; con questa opzione i programmi saranno leggermente meno veloci, in quanto il BASIC dovrà salvare e ripristinare lo stato della memoria EMS prima e dopo tutte le chiamate alle routine nelle Quick Library (tra cui INTERRUPT e INTERRUPTX). Questo rallentamento influisce però solo sui programmi interpretati, e non quelli compilati.

Prima di utilizzare la memoria espansa un programma deve accertarsi che il sistema su cui è eseguito disponga effettivamente di memoria espansa, ossia che sia stato caricato al bootstrap un device driver per la gestione della memoria EMS e che tale driver sia funzionante. Lo standard EMS prevede che la comunicazione tra i programmi applicativi e il driver avvenga attraverso l'interrupt software 67 (esadecimale), un po' come il colloquio del Dos avviene attraverso l'interrupt 21h.

Purtroppo non possiamo usare i servizi offerti dal driver per capire se il driver è realmente installato. Il motivo è semplice: se il driver non è installato, esso non può notificare al programma che esso stesso non esiste ! Non solo: sebbene tutti i BIOS più recenti adottino la precauzione di far puntare tutti i vettori di interrupt inutilizzati dal Dos ad una istruzione IRET (*return from interrupt*), molti dei primi BIOS erano molto meno educati, e richiamare l'interrupt 67h prima di testare che esista effettivamente un driver in grado di soddisfare la richiesta può portare, su alcuni sistemi, al blocco totale del sistema. Occorre quindi un sistema per testare l'esistenza del driver EMS prima di richiedere una qualsiasi funzione del driver stesso. Esistono almeno due metodi per raggiungere lo scopo: il criterio consigliato dallo standard EMS ed un secondo metodo non standard ma più veloce del primo ed altrettanto affidabile.

Il metodo consigliato consiste nel tentare di aprire un file di nome "EMMXXXX0"; tutti i device driver che gestiscono la memoria espansa si registrano presso il sistema operativo con questo nome (esattamente come

la console è registrata come il device CON, la stampante come PRN, ecc.), per cui se il tentativo fallisce si può essere certi che il driver EMS non esiste. D'altra parte, se il tentativo riesce vi è una piccola ma non trascurabile probabilità che esista nella directory corrente un file con quel nome, e per escludere tale possibilità occorre eseguire un servizio IOCTL del Dos che ci permetta di stabilire se l'handle restituito dall'operazione di OPEN si riferisce ad un file o un device. In tutti i casi, al termine del controllo occorre chiudere il file o il device associato all'handle, in modo da poter riutilizzare l'handle nel corso del programma (per inciso, questo è il metodo usato implicitamente dal programma batch visto in precedenza):

```
FUNCTION IsEmsThere%
  '______________________________
  ' Controlla l'esistenza di un driver EMS
  ' seguendo il metodo consigliato dallo standard
  '______________________________
  DIM reg AS regType
  ' tenta di aprire un device di nome "EMMXXXX0"
  ON LOCAL ERROR GOTO FileNotFound
  OPEN "EMMXXXX0" FOR INPUT AS #1
  ' se non vi è errore occorre controllare che si
  ' tratti di un device e non di un file, chiamando un
  ' particolare servizio del Dos
  reg.bx = FILEATTR(1, 2)     ' handle del file in BX
  reg.ax = &h4400        ' servizio 44 sottoservizio 00
  Interrupt &H21, reg, reg    ' richiama il Dos
  IF reg.dx AND 128 THEN    ' testa il bit 7 di DL
  IsEmsThere% = -1    ' se non zero è un device, quindi
  END IF          ' esiste un driver per EMS
  FileNotFound:           ' si arriva a questo punto anche
  EXIT FUNCTION         ' se la OPEN è fallita
  END FUNCTION
```

Il metodo alternativo consiste nel recuperare l'indirizzo del vettore 67h, ossia il punto di ingresso della routine invocata quando si esegue una istruzione INT 67, e verificare che all'offset 10 del blocco di memoria a cui punta il vettore di interrupt si trovi la stringa "EMMXXXX0", che identifica il device driver come un gestore di memoria espansa. Questo metodo evita l'overhead derivante dalla apertura e chiusura dei file, ed inoltre è l'unico modo per testare la presenza di un driver EMS dall'interno di un altro device driver oppure di un interrupt handler:

```
FUNCTION IsEmsThere2%
  '______________________________
  ' Controlla l'esistenza di un driver EMS
  ' ispezionando direttamente il segmento a cui punta l'int 67h
  '______________________________
  DIM addr%, segment%, offset%, temp$
  ' determina il segomento a cui punta il vettore 67h; questo
  ' valore è conservato in bassa memoria all'indirizzo 67h*4+2
  DEF SEG = 0
  addr% = &H67 * 4 + 2
  segment% = CVI(CHR$(PEEK(addr%)) + CHR$(PEEK(addr% + 1)))
  ' leggi la stringa lunga 8 caratteri, all'offset 10
```

```
    DEF SEG = segment%
    FOR offset% = 10 TO 17
      temp$ = temp$ + CHR$(PEEK(offset%))
    NEXT
    DEF SEG
    ' confronta con l'identificativo del driver
    IsEmsThere2% = (temp$ = "EMMXXXX0")
    END FUNCTION
```

Dopo aver controllato la presenza del driver per la gestione della memoria espansa, il nostro programma deve accertarsi che il driver funzioni a dovere, e che non abbia rilevato alcuna anomalia hardware nella memoria stessa: questa operazione, consigliata caldamente dalle specifiche EMS e indispensabile quando la memoria è situata su una scheda separata, nel caso della memoria espansa simulata dai processori 386 e 486 ha molta meno importanza, ma poiché si tratta di poche istruzioni possiamo benissimo includerla nei nostri programmi:

```
FUNCTION CheckEmsDriver%
  '______________________________
  ' Controlla che la memoria EMS sia funzionante
  ' restituisce 0 se è tutto OK, oppure il codice di errore
  '______________________________
  DIM reg AS RegType
  reg.ax = &H4000       ' servizio 40h
  Interrupt &H67, reg, reg    ' chiama il driver EMS
  ' per isolare il valore in AH è necessario forzare
  ' le operazioni a 32 bit
  CheckEmsDriver% = (reg.ax AND &HFF00&) \ 256
END FUNCTION
```

La routine precedente mostra una tipica chiamata al driver EMS; in ingresso il registro AH deve contenere il numero del servizio richiesto, AL deve contenere il numero del sotto-servizio (nell'esempio precedente non è stato necessario); al ritorno dall'interrupt, il registro AH contiene zero se la chiamata ha avuto successo, oppure un codice di errore. I codici di errore sono codificati a partire da 80h; tutti i possibili codici di errore sono riportati nelle tabelle riportate alla fine di questo capitolo.

Dopo aver controllato che il driver EMS è installato e funzionante, è opportuno verificare il suo numero di versione, in modo da evitare delle chiamate non supportate dal gestore stesso. Il servizio che ci interessa è il 46h, che restituisce in AL il numero di versione codificato in BCD (Binary Code Decimal), con la porzione intera nei bit 4-7 e il numero decimale nei bit 0-3; ad esempio, la versione 3.2 restituisce in AL il valore 32h. Ecco la routine BASIC che calcola correttamente il numero di versione del driver:

```
FUNCTION EmsVersion%
'______________________________
' Restituisce il numero di versione del driver EMS
' moltiplicato per 100 (ad es. 320 per la versione 3.2)
'______________________________
  DIM reg AS RegType, major%, minor%
```

```
    reg.ax = &H4600        ' servizio 46h
    Interrupt &H67, reg, reg
    ' il "major release" nei bit 4-7 di AL
    major% = (reg.ax AND &HF0) \ 16
    ' il "minor release" nei bit 0-3 di AL
    minor% = (reg.ax AND &HF)
    ' componi il risultato
    EmsVersion% = major% * 100 + minor% * 10
  END FUNCTION
```

ALLOCAZIONE DELLA MEMORIA ESPANSA

Dopo aver controllato il numero di versione, possiamo finalmente richiedere di allocare una o più pagine di memoria EMS, ciascuna delle quali è lunga 16K. Attenzione quindi ad arrotondare per eccesso il risultato della divisione:

numero Kbyte necessari / 16

ad esempio 32K richiedono due pagine logiche, mentre 33K ne richiedono tre; attenti anche a non passare un numero di pagine pari a zero. Il servizio che alloca il numero richiesto di pagine è il servizio 43h; se non vi sono errori, il driver EMS restituisce in DX l'handle delle pagine appena allocate; tale handle è un numero che ci permetterà di effettuare in seguito tutte le operazioni necessarie sulla zona di memoria EMS appena allocata, ed è molto simile agli handle restituiti dal sistema operativo quando si apre un file.

```
SUB EmsAllocate (kbytes%, handle%, errorCode%)
  '__________________________
  ' Alloca il numero desiderato di kbytes
  ' restituisce l'handle e il codice di errore
  '__________________________
  DIM reg AS RegType
  reg.ax = &H4300        ' servizio 43h
  reg.bx = (kbytes% + 15) \ 16  ' pagine richieste in BX
  Interrupt &H67, reg, reg    ' chiama il driver EMS
  errorCode% = (reg.ax AND &HFF00&) \ 256
  IF errorCode% = 0 THEN     ' se non vi è errore
  handle% = reg.dx     ' l'handle è restituito in DX
  END IF
END FUNCTION
```

Ecco come usare la procedura EmsAllocate

```
' alloca 120K in memoria espansa
EmsAllocate 120, handle%, errorCode%
IF errorCode% THEN
PRINT "Impossibile allocare la memoria: errore #"; errorCode%
END IF
```

In condizioni "normali" (cioè se non vi sono malfunzionamenti del driver) il servizio di allocazione può restituire quattro differenti errori:

85H nessun handle disponibile

87H richieste più pagine di quante fisicamente presenti

88H richieste più pagine del numero di pagine correntemente libere

89H richieste zero pagine

Cosa accade se la richiesta di allocazione fallisce con gli errori 87H o 88H ? In tal caso il programmatore deve decidere se non utilizzare affatto la memoria EMS oppure cercare di arrangiarsi con la memoria effettivamente a disposizione. Per conoscere quante sono le pagine libere si utilizza il servizio 42H, che restituisce anche il numero totale di pagine (libere o già allocate ad un programma):

```
SUB EmsPages (totalPages%, freePages%)
  '___________________________
  ' Restituisce il numero di pagine EMS complessive
  '   e il numero delle pagine ancora disponibili (libere)
  '___________________________
  DIM reg AS RegType
  reg.ax = &H4200        ' servizio 42h
  Interrupt &H67, reg, reg    ' chiama il driver EMS
  ' (non è necessario testare il codice di errore)
  ' il numero delle pagine totali è restituto in DX
  totalPages% = reg.dx
  ' il numero delle pagine libere è restituito in BX
  freePages% = reg.bx
END SUB
```

La procedura EmsPages potrebbe anche essere utile in un programma diagnostico, che mostra l'ammontare della memoria EMS installata e libera, oppure in un programma di installazione "intelligente". E' facile mettere insieme i servizi 42h e 43h per scrivere una routine che alloca tutta la memoria EMS libera:

```
SUB EmsAllocateAll (kbytes%, handle%, errorCode%)
  '___________________________
  ' Alloca tutta la memoria EMS libera
  ' restituisce il numero di Kbyte allocati, l'handle
  '   e il codice di errore
  '___________________________
  DIM reg AS RegType
  reg.ax = &H4200        ' servizio 42h
  Interrupt &H67, reg, reg  ' chiama il driver EMS
  ' il numero delle pagine libere è restituito in BX
  kbytes% = reg.bx * 16
  reg.ax = &H4300        ' servizio 43h
  Interrupt &H67, reg, reg  ' chiama il driver EMS
  errorCode% = (reg.ax AND &HFF00&) \ 256
  IF errorCode% = 0 THEN    ' se non vi è errore
    handle% = reg.dx     ' l'handle è restituito in DX
  END IF
END FUNCTION
```

La procedura EmsAllocateAll ha anche un utilizzo abbastanza inconsueto, quello di allocare tutta la memoria EMS libera e di sottrarla i programmi eseguiti in seguito: in questo modo potremo testare il comportamento di un programma applicativo in assenza di memoria espansa, senza dover modificare il file CONFIG.SYS e resettare il sistema per disattivare il driver EMS.

Troppi programmatori testano i propri programmi sui loro sistemi (tipicamente dei 386 o 486 con 4-8 megabyte di RAM e disco rigido da 12 millisecondi) e si dimenticano di considerare che il loro programma sarà con molta probabilità eseguito su semplici 8086 o 80286. E' buona norma, invece, testare il comportamento del programma su macchine inferiori e in situazioni limite, usando quando è possibile tecniche come quella appena descritta.

Dopo aver allocato una o più pagine in memoria espansa, il passo successivo è ovviamente quello di scrivere dei dati in questa zona di memoria. Come sappiamo, la memoria espansa è situata al di fuori dello spazio di indirizzamento, per cui non è possibile leggere e scrivere in quelle locazioni. L'unico modo è di richiedere al driver di eseguire il mapping di una determinata pagina logica in una delle quattro pagine fisiche a disposizione nel frame buffer. Ma dove si trova questo frame buffer ? Per saperlo occorre richiamare ancora una volta un servizio del driver:

```
FUNCTION EmsFrameBuffer%
  '-----------------------
  ' Restituisce il segmento del frame buffer
  '-----------------------
  DIM reg AS RegType
  reg.ax = &H4100        ' servizio 41h
  Interrupt &H67, reg, reg
  ' il segmento del frame buffer è restituito in BX
  EmsFrameBuffer% = reg.bx
END FUNCTION
```

Si noti che il frame buffer è sempre allineato al paragrafo, per cui è sufficiente conoscere il segmento per poterlo individuare. Supponiamo che il frame buffer si trovi al segmento D000; allora questo sarà l'indirizzo della pagina fisica 0, mentre la pagina fisica 1 si troverà 16K più in là al segmento D400, la pagina fisica 2 al segmento D800 e la pagina fisica 3 a DC00.

IL MAPPING DELLE PAGINE LOGICHE

Siamo finalmente pronti a richiamare la pagina logica che ci interessa in una qualsiasi delle quattro pagine fisiche, utilizzando il servizio 44h; tale servizio richiede in DX l'handle restituito dalla funzione di allocazione, in AL il numero della pagina fisica (nell'intervallo 0-3), in BX il numero della pagina logica (compresa tra zero e N-1, dove N è il numero di pagine allocate all'handle):

```
SUB EmsMapPage (handle%, physical%, logical%)
  '-----------------------
  '  Esegue il mapping di una pagina logica in unadelle
  '  quattro pagine fisiche
  '-----------------------
  DIM reg AS RegType
  ' il numero del servizio in AH, il numero della pag.fisica in AL
```

```
    reg.ax = &H4400 + (physical% AND 3)
    ' il numero della pagina logica in BX
    reg.bx = logical%
    ' il valore dell'handle in DX
    reg.dx = handle%
    Interrupt &H67, reg, reg
  END SUB
```

A meno di grossolani errori di programmazioni riguardo il numero della pagina fisica e logica o dell'handle, questo servizio non dovrebbe mai provocare errore.

Effettuato con successo il mapping, non ci rimane che usare in qualche modo la pagina logica che si trova finalmente nello spazio di indirizzamento del microprocessore. Poiché il BASIC è sprovvisto di puntatori, è possibile accedere direttamente all'area in questione solo attraverso una serie di PEEK e POKE. Ecco un esempio di come è possibile allocare una pagina da 16K in memoria espansa e conservarvi il contenuto di una stringa:

```
' alloca una pagina in memoria espansa
EmsAllocate 16%, handle%, errorCode%
IF errorCode% THEN PRINT "Errore!!!": END
' mapping della pagina logica 0 nella pagina fisica 0
EmsMapPage handle%, 0, 0
' rendi accessibile la prima pagina fisica con PEEK/POKE
DEF SEG = EmsFrameBuffer%
' copia il contenuto della stringa info$ - notare che
' l'offset parte da zero, mentre l'indice parte da uno
length% = LEN(info$)
FOR i% = 1 TO length%
  POKE i% - 1, ASC(MID$(info$, i%))
NEXT
DEF SEG
```

Quando in seguito sarà necessario recuperare i dati si procederà come segue:

```
EmsMapPage handle%, 0, 0
DEF SEG = EmsFrameBuffer%
info$ = SPACE$(length%)
FOR i% = 1 TO length%
  MID$(info$, i%, 1) = CHR$(PEEK(i% - 1))
NEXT
DEF SEG
```

Più avanti vedremo come sfruttare i servizi della versione 4.0 del driver per copiare direttamente i dati dalla memoria espansa ad una particolare locazione in memoria convenzionale, e viceversa.

Se vogliamo ottimizzare le prestazioni dei nostri programmi possiamo eseguire il mapping su tutte le quattro pagine fisiche, in modo da poter lavorare con blocco di 64K

```
EmsMapPage handle%, 0, 0
EmsMapPage handle%, 1, 1
EmsMapPage handle%, 2, 2
EmsMapPage handle%, 3, 3
' carica dei dati da file usando BLOAD
DEF SEG = EmsFrameBuffer%
BLOAD "schermo.dat", 0
```

Anche se finora abbiamo usato le quattro pagine fisiche per eseguire il mapping di altrettante pagine logiche consecutive e relative allo stesso handle, nulla ci vieta di fare riferimento a pagine logiche qualsiasi, incluso pagine logiche allocate ad handle differenti. Un esempio concreto potrebbe essere un programma per il calcolo su matrici di grandi dimensioni, per cui potremmo stabilire di allocare un handle per matrice, per poi processarle 16K alla volta. La decisione di allocare tutta la memoria richiesta ad un unico handle oppure suddividere la richiesta tra più handle spetta solo al programmatore, ma si tenga presente che la memoria espansa può essere usata solo in blocchi di 16K, il che può provocare sprechi indesiderati: ad esempio dieci blocchi di 20K richiedono ben venti pagine, anziché le tredici che sarebbero sufficienti allocando un unico blocco di 200K.

DEALLOCAZIONE DELLA MEMORIA

Abbiamo già accennato al fatto che tutta la memoria espansa allocata dal programma deve essere rilasciata prima di ritornare al sistema operativo. In questo aspetto la memoria espansa si differenzia dalla memoria convenzionale che è automaticamente deallocata dal Dos quando il programma termina: se un programma non dealloca la memoria espansa utilizzata, essa sarà sottratta agli altri programmi, fino al prossimo reset del sistema.

Il servizio 45h provvede a rilasciare la memoria, e richiede che il registro DX sia stato caricato con l'handle:

```
SUB EmsDeallocate (handle%)
  '______________________________
  ' Dealloca tutta la memoria EMS associata a un handle
  '______________________________
  DIM reg AS RegType
  reg.ax = &H4500        ' servizio 45h
  reg.dx = handle%        ' handle in DX
  Interrupt &H67, reg, reg
END SUB
```

Ovviamente questo servizio deve essere richiamato per ciascuno degli handle allocati al programma. Se il programma è corretto, per cui l'handle passato in DX corrisponde effettivamente a della memoria EMS allocata in precedenza dallo stesso programma, questo servizio non dovrebbe mai provocare errore.

Poiché la memoria EMS non è rilasciata automaticamente uscendo dal programma, nella fase di test è possibile che il programma termini in maniera inaspettata a causa di un errore, e sarà necessario resettare il sistema per ripartire con tutta la memoria disponibile. Inoltre, se il programma può essere interrotto dalla pressione dei tasti Ctrl-C o Ctrl-Break oppure da un errore critico, il programmatore dovrebbe prendere le dovute precauzioni per evitare

che tale uscita inaspettata lasci delle porzioni di memoria EMS allocate e di fatto non disponibili agli altri programmi.

LO STANDARD EMS 4.0

Esistono in realtà tre standard EMS distinti, individuati dai loro numeri di versione 3.0, 3.2 e 4.0. Anche se i vari manuali consigliano, se possibile, di attenersi ai servizi forniti dal primo 3.0, in realtà tutti i manager di memoria espansa in circolazione sono allineati alla più recente versione 4.0 (che poi tanto recente non è in quanto risale al 1987). Di seguito vedremo come avvantaggiarsi di alcune delle possibilità offerte dallo standard 4.0:

- le pagine fisiche e logiche possono essere di qualunque dimensione, e non sono limitate ai 16K
- le pagine fisiche possono essere "mappate" in memoria a qualunque indirizzo
- la quantità totale di memoria gestibile è di 32 Megabyte (contro gli 8 megabyte delle versioni precedenti)
- nuovi servizi per trasferire memoria dalla memoria EMS alla memoria convenzionale e viceversa, per eseguire il mapping di più pagine per volta, di proteggere delle pagine dal reset "a caldo" di sistema
- numerosi altri servizi concepiti per essere sfruttati dal sistema operativo

Dal punto di vista del programmatore, la maggior parte dei nuovi servizi dello standard 4.0 sono superflui, nel senso che sono inutili o raramente usati (ad es. la possibilità di proteggere la memoria dal reboot); la caratteristica forse più importante, la possibilità di gestire fino a 32M di memoria, può essere sfruttata senza richiamare esplicitamente un servizio del driver, e quindi senza sacrificare la compatibilità con le versioni precedenti dello standard.

Dal punto di vista del programmatore BASIC, uno dei servizi più interessanti dello standard 4.0 permette di trasferire direttamente il contenuto di una zona di memoria convenzionale in una pagina logica EMS o viceversa, senza curarsi di ottenere l'indirizzo del frame buffer, di eseguire il mapping delle pagine richieste e di eseguire "manualmente" il trasferimento dei dati con delle PEEK e POKE. Il servizio è il 57h, sottoservizio 00, e richiede che il programmatore abbia preparato un buffer contenente tutte le informazioni necessarie al trasferimento dei dati; l'indirizzo di tale buffer deve essere passato al driver EMS nella coppia di registri DS:SI e i dati nel buffer devono essere preparati con il formato seguente:

offset	formato	descrizione
0	DWORD	lunghezza in byte
4	BYTE	sorgente della copia (0=convenzionale, 1=ems)
5	WORD	handle sorgente (zero se memoria convenzionale)
7	WORD	offset del sorgente
9	WORD	segmento del sorgente (se memoria convenzionale)
10	WORD	pagina fisica del sorgente (se memoria EMS)
11	BYTE	destinazione della copia (0=convenzionale, 1=ems)
12	WORD	handle destinazione(zero se memoria convenzionale)
14	WORD	offset della destinazione
16	WORD	segmento della destinazione (se mem.convenzionale)
		pagina fisica della destinazione (se memoria EMS)

Conviene allora definire una struttura TYPE...END TYPE e scrivere due procedure BASIC che usino questo servizio per trasferire dati dalla memoria convenzionale alla memoria espansa, e viceversa:

```
TYPE EmsMoveType
  length        AS LONG
  sourceType    AS STRING * 1
  sourceHandle  AS INTEGER
  sourceOffset  AS INTEGER
  sourceSegment AS INTEGER
  destType      AS STRING * 1
  destHandle    AS INTEGER
  destOffset    AS INTEGER
  destSegment   AS INTEGER
END TYPE
SUB MoveMemoryToEms (segment%, offset%, handle%, page%, emsOffset%, bytes&)
  '________________________________
  ' Sposta un'area dalla memoria convenzionale alla memoria EMS
  '________________________________
  DIM reg AS RegTypeX, emsMove AS EmsMoveType
  ' non è necessario inizializzare i campi sourceType e sourceHandle
  emsMove.length = bytes&
  emsMove.sourceOffset = offset%
  emsMove.sourceSegment = segment%
  emsMove.destType = CHR$(1)
  emsMove.destHandle = handle%
  emsMove.destOffset = emsOffset%
  emsMove.destSegment = page%
  ' l'indirizzo della struttura in DS:SI
  reg.ds = VARSEG(emsMove)
  reg.si = VARPTR(emsMove)
  ' numero servizio 57h in AH, sottoservizio 00 in AL
  reg.ax = &H5700
  InterruptX &H67, reg, reg
END SUB

SUB MoveMemoryFromEms (handle%, page%, emsOffset%, segment%,offset%,bytes&)
  '________________________________
  ' Sposta un'area dalla memoria EMS alla memoria convenzionale
  '________________________________
  DIM reg AS RegTypeX, emsMove AS EmsMoveType
  ' non è necessario inizializzare i campi destType e destHandle
  emsMove.length = bytes&
  emsMove.sourceType = CHR$(1)
  emsMove.sourceHandle = handle%
  emsMove.sourceOffset = emsOffset%
  emsMove.sourceSegment = page%
```

```
    emsMove.destOffset = offset%
    emsMove.destSegment = segment%
    ' l'indirizzo della struttura in DS:SI
    reg.ds = VARSEG(emsMove)
    reg.si = VARPTR(emsMove)
    ' numero servizio 57h in AH, sottoservizio 00 in AL
    reg.ax = &H5700
    InterruptX &H67, reg, reg
END SUB
```

il seguente frammento di codice mostra come copiare 128K di memoria convenzionale all'indirizzo B800:0500 in memoria espansa, a partire dalla pagina logica uno dell'handle 5:

```
MoveMemoryToEms &HB800, &H500, 5, 1, 0, 131072
```

Con queste procedure è possibile trasferire fino a un megabyte di dati; se la lunghezza dell'area da copiare supera quella di una singola pagina logica sono influenzate tutte le pagine logiche successive necessarie. Come si vede dalla tabella alla fine di questo capitolo, molti dei codici di errore introdotti con lo standard 4.0 si riferiscono proprio a questo servizio; si noti che se una delle due aree è in memoria convenzionale e la lunghezza dell'area è tale da provocare un *wrapping* degli indirizzi (ossia l'indirizzo iniziale più la lunghezza supera un megabyte) il servizio restituisce un errore di codice A2H.

Il sottoservizio 01 del servizio 57h è strettamente imparentato con il precedente, in quanto permette di *scambiare* il contenuto di due aree di memoria, che possono trovarsi indifferentemente in memoria convenzionale o espansa; il formato del buffer è identico a quello visto in precedenza. E' facile quindi modificare una delle procedure precedenti per eseguire lo scambio di dati tra la memoria convenzionale e la memoria espansa:

```
SUB ExchangeEmsMemory (segment%, offset%, handle%, page%,emsOffset%,bytes&)
    '-----------------------------------------------
    ' Scambia il contenuto di un'area in memoria convenzionale
    ' con una area in memoria EMS
    '-----------------------------------------------
    DIM reg AS RegTypeX, emsMove AS EmsMoveType
    ' non è necessario inizializzare i campi sourceType e sourceHandle
    emsMove.length = bytes&
    emsMove.sourceOffset = offset%
    emsMove.sourceSegment = segment%
    emsMove.destType = CHR$(1)
    emsMove.destHandle = handle%
    emsMove.destOffset = emsOffset%
    emsMove.destSegment = page%
    ' l'indirizzo della struttura in DS:SI
    reg.ds = VARSEG(emsMove)
    reg.si = VARPTR(emsMove)
    ' numero servizio 57h in AH, sottoservizio 01 in AL
    reg.ax = &H5701
    InterruptX &H67, reg, reg
END SUB
```

nel caso in cui le due aree si sovrappongono lo scambio non avviene e viene restituito un errore con codice 97H.

INFORMAZIONI SUL DRIVER EMS

Il gestore di memoria EMS supporta anche un certo numero di funzioni diagnostiche, che restituiscono informazioni sul gestore stesso e sulle varie zone di memoria allocate o ancora libere; questi servizi possono essere usati da un programma diagnostico, oppure sfruttati dall'interno dei nostri stessi programmi con funzioni di debug. I servizi descritti di seguito sono comuni alle tre versioni dello standard EMS.

Il servizio 4Bh restituisce in BX il numero degli handle correntemente impegnati (a cui cioè sono associate delle pagine di memoria EMS);

```
FUNCTION EmsActiveHandles%
  '______________________________
  ' Restituisce il numero degli handle attivi
  '______________________________
  DIM reg AS RegType
  reg.ax = &H4B00
  Interrupt &H67, reg, reg
  EmsActiveHandles% = reg.bx
END FUNCTION
```

Il servizio 4CH è utile per determinare il numero di pagine logiche associate ad un particolare handle:

```
FUNCTION EmsAllocatedPages% (handle%)
  '______________________________
  ' Restituisce il numero di pagine associate a un handle
  '______________________________
  DIM reg AS RegType
  reg.ax = &H4C00
  reg.dx = handle%
  Interrupt &H67, reg, reg
  EmsAllocatedPages% = reg.bx
END FUNCTION
```

Per ottenere le informazioni su tutti gli handle in un'unica operazione si può usare il servizio 4DH, che restituisce in un buffer a cui punta ES:DI una serie di coppie di word nel formato (handle, numeroPagine) e il numero di handle attivi in BX; poiché vi possono essere fino a 256 handle attivi il buffer dovrebbe essere lungo almeno 1K (256 x 4 byte):

```
SUB EmsHandleInfo (numHandles%, buffer%())
  '______________________________
  ' Restituisce il numero di pagine associate a ciascun handle
  '______________________________
  DIM reg AS RegTypeX
  reg.ax = &H4D00
  reg.es = VARSEG(buffer%(0))
  reg.di = VARPTR(buffer%(0))
  InterruptX &H67, reg, reg
  numHandles% = reg.bx
END SUB
```

Ecco come usare la procedura **EmsHandleInfo** in un programma:

```
' dimensionare un vettore di almeno 512 elementi
DIM buffer%(511)
```

```
EmsHandleInfo numHandles%, buffer%()
PRINT "HANDLE#" TAB(20);"PAGINE ASSOCIATE"
FOR i% = 1 TO numHandles%
  PRINT buffer%(i% * 2); TAB(20); buffer%(i% * 2 + 1)
NEXT
```

A questo punto avete a disposizione tutte le informazioni per creare programmi BASIC in grado di sfruttare al meglio la memoria oltre il primo megabyte.

Codici di errore restituiti dai driver 3.0, 3.2 e 4.0

00H nessun errore

80H errore interno (probabile corruzione della memoria del driver)

81H malfunzionamento hardware

82H manager della memoria già attivo

83H handle non valido

84H numero di funzione non definito

85H Errore di allocazione: nessun handle disponibile

86H errore nel salvataggio o nel ripristino del contesto di mapping

87H Errore di allocazione: richieste più pagine di quante installate nel sistema

88H Errore di allocazione: richieste più pagine del numero di pagine libere

89H Errore di allocazione: impossibile allocare zero pagine

8AH Errore di mapping: numero di pagina logica fuori dall'intervallo ammesso per l'handle

8BH Errore di mapping: numero di pagina fisica fuori dall'intervallo 0-3

8CH L'area per salvare le informazioni del page-mapping è piena

8DH Errore nel salvataggio del contesto di mapping: l'area di salvataggio già contiene informazioni di contesto relative allo stesso handle

8EH Errore nel ripristino del contesto di mapping: l'area di salvataggio non contiene informazioni di contesto associate all'handle

8FH Numero di sottofunzione non definito

Codici di errore restituiti dal solo driver 4.0

90H tipo di attributo non definito

91H caratteristica non supportata

92H le aree sorgenti e destinazione hanno lo stesso handle e si sovrappongono; il trasferimento è avvenuto ma parte della regione sorgente è stata sovrascritta

93H la lunghezza specificata per l'area sorgente o destinazione è maggiore della quantità di memoria effettivamente allocata

94H l'area di memoria convenzionale e l'area di memoria espansa si sovrappongono

95H l'offset specificato è fuori dall'intervallo permesso

96H la lunghezza specificata è maggiore di 1 megabyte

97H le aree sorgenti e destinazione hanno lo stesso handle e si sovrappongono; lo scambio di memoria non può essere effettuato

98H il tipo di memoria dell'area sorgente o destinazione non è definito (diverso da zero e 1)

99H (codice di errore non usato)

9AH il driver supporta i set di registri map o DMA alternativi, ma il set specificato non è supportato

9BH il driver supporta i set di registri map o DMA alternativi, ma tutti i set alternativi sono correntemente allocati

9CH il driver non supporta i set di registri map o DMA alternativi e si è tentato di specificare un set di registri non nullo

9DH il driver supporta i set di registri map o DMA alternativi, ma il set alternativo di registri specificato non è definito oppure non è stato allocato

9EH il driver non supporta i canali DMA dedicati

9FH il driver supporta i canali DMA dedicati, ma il canale DMA specificato non è supportato

A0H nessun handle trovato per il nome specificato

A1H esiste già un handle con questo nome

A2H l'indirizzo di memoria iniziale sommato alla lunghezza supera il megabyte

A3H pointer non valido, oppure il contenuto dell'array sorgente è stato danneggiato

A4H accesso alla funzione negato dal sistema operativo

7

TECNICHE DI OTTIMIZZAZIONE

PROGRAMMI PIU' VELOCI CON 'INTEGER' E 'LONG'

È noto che le operazioni che coinvolgono gli interi sono molto molto più veloci delle operazioni sui numeri reali. I risultati di un benchmark informale mostrano i calcoli effettuati con variabili SINGLE o DOUBLE sono da cinquanta (nel caso della moltiplicazione) a sessanta volte (addizione e sottrazione) più lenti che nel caso di numeri INTEGER. Le addizioni con i numeri LONG sono praticamente altrettanto veloci che quelle sugli interi a 16 bit, mentre le moltiplicazioni sono almeno quattro volte più lente: la causa di questa apparente contraddizione è dovuta al fatto che bastano due sole istruzioni Assembly per sommare due numeri a 32 bit, mentre ne servono qualche decina per una moltiplicazione: viceversa, per gli interi a 16 bit basta sempre e comunque una sola istruzione Assembly, anche se nel caso della moltiplicazione essa impiega un numero di cicli di clock più alto (a dire il vero, sui processori 386 e 486 è possibile effettuare tutte le operazioni a 32 bit con una sola istruzione, ma le versioni del linguggio fino al BASIC PDS 7.1 non sono in grado di generare istruzioni per questi processori).

Il formato CURRENCY è stato introdotto con il BASIC PDS 7.0 per facilitare i calcoli di tipo gestionale su cifre molto grandi, con un tipo di memorizzazione chiamato in gergo *fixed decimal*, in cui vi sono sempre quattro cifre decimali (per contrapposizione i numeri reali si memorizzano in un formato chiamato *floating point*, a virgola fluttuante). I numeri in questo formato, che coinvolge 64 bit, si comportano molto bene nelle addizioni (in cui sono veloci quasi quanto i

numeri interi a 16 bit), mentre nelle moltiplicazioni e nelle divisioni sono persino più lenti dei numeri in formato DOUBLE. Anche in questo caso il risultato si spiega col fatto che sono sufficienti solo otto istruzioni Assembly per sommare o sottrarre due numeri in questo formato, mentre occorrono anche centinaia di istruzioni per le più complesse moltiplicazioni e divisioni.

In definitiva, quando è possibile scegliere, le operazioni con gli interi sono sempre da preferirsi a quelle che coinvolgono i numeri reali. Una buona abitudine consiste nell' inserire all'inizio di ogni programma l'istruzione:

```
DEFINT A-Z
```

per default tutte le variabili BASIC sono di tipo SINGLE, ma con l'istruzione precedente ci assicuriamo che il tipo di default sia INTEGER. Attenzione, però, perché non è sufficiente usare variabili intere per ottenere la massima velocità: è anche necessario che l'operazione non produca risultati intermedi in virgola mobile. Facciamo un esempio:

```
FOR i = 1 TO 10000
        array(i) = i / 2
NEXT
```

Nel caso precedente, sebene tutte le quantità in gioco siano intere, il risultato della divisione non lo è, e il BASIC correttamente lo calcola in doppia precisione, anche se poi lo tronca per poterlo assegnare ad un elemento di un vettore di interi. Il problema si risolve usando l'operatore "\", che effettua una divisione intera

```
FOR i = 1 TO 10000
        array(i) = i \ 2
NEXT
```

Molte istruzioni BASIC restituiscono un valore reale: a parte ovviamente le funzioni trigonometriche e le funzioni SQRT e EXP, occorre tenere d'occhio la funzione VAL. Ad esempio, il seguente programma non è molto efficiente:

```
FOR i = 1 TO 10000
        array%(i) = VAL(a$)
NEXT
```

in quanto l'espressione nel corpo del ciclo è calcolata come valore in virgola mobile; è meglio riscrivere il programma in questo modo:

```
temp% = VAL(a$)
FOR i = 1 TO 10000
        array%(i) = temp%
NEXT
```

Occorre fare attenzione al fatto che per il BASIC i confronti non sono che particolari operazioni aritmetiche, che restituiscono 0 oppure -1; questo significa che il discorso fatto a proposito della convenienza nell'usare gli interi si estende anche agli operatori di confronto. Per esempio

```
IF x% > CINT(VAL(a$)) THEN
```

è decisamente più efficiente di

```
IF x% > VAL(a$) THEN
```

perché il compilatore può sapere in anticipo che il confronto avviene tra due interi; si sarebbe tentati dall'usare la funzione INT anziché CINT, ma non si otterrebbe il risultato sperato: la funzione INT può anche restituire un valore LONG o addirittura SINGLE o DOUBLE.

EVITARE CALCOLI IN VIRGOLA MOBILE

Si tratta di un corollario di quanto detto nel paragrafo precedente, ma in questo caso l'accento va posto non tanto sulla velocità del programma quanto sulla dimensione del codice prodotto. Se il programma contiene anche un solo calcolo in virgola mobile, il linker include l'intero package per l'aritmetica con numeri reali, aumentando le dimensioni del programma di 10K o più.

Quindi se intendete creare eseguibili di piccole dimensioni vale quanto detto sull'uso degli interi (questo è un caso in cui le tecniche per ottimizzare la velocità di esecuzione coincidono con quelle per minimizzare le dimensioni del programma). In particolare occorre stare molto attenti alla divisione floating "/" e ad una serie di funzioni e istruzioni BASIC che utilizzano numeri reali: DRAW, PMAP, POINT, PRINT USING, INPUT, RANDOM, READ, RND, SOUND, TIMER, VAL e WINDOW.

Per esempio, la funzione TIMER può essere in molti casi sostituita da una funzione che ricava il numero di *ticks* (18-esimi di secondo) trascorsi dalla mezzanotte direttamente dall'area di memoria usata dal BIOS, ed è anche possibile creare un comando **Pause** che sospende l'elaborazione del programma per il numero indicato di tick:

```
FUNCTION SystemTicks&
      '====================================================
      ' Il numero di ticks trascorsi da mezzanotte
      '====================================================
      ' il valore che ci interessa è memorizzato in quattro byte
      ' consecutivi a partire dall'indirizzo 0000:046C
      DEF SEG = 0
      SystemTicks& = PEEK(&H46C) + &H100& * PEEK(&H46D) + &H10000& * _
              PEEK(&H46E) + &H1000000 * PEEK(&H46F)
      DEF SEG
END FUNCTION
SUB Pause (ticks%)
      '==========================================================
      ' Sospende il programma per il numero indicato di ticks
      '==========================================================
      DIM counter%, byte%
      DEF SEG = 0
      ' ad ogni iterazione attende che il byte all'indirizzo
      ' 0000:046C cambi valore
      FOR counter% = 1 TO ticks%
              byte% = PEEK(&H46C)
              DO: LOOP WHILE PEEK(&H46C) = byte%
```

```
        NEXT
        DEF SEG
    END SUB
```

E' anche abbastanza semplice scrivere una funzione alternativa a VAL che lavora solo con numeri interi, in modo da poter usare LINE INPUT invece di INPUT:

```
    FUNCTION ValInt& (a$)
        '==========================================================
        ' Estrae un valore intero da una stringa
        '==========================================================
        DIM index%, result&, sign%, start%
        IF LEN(a$) = 0 THEN ValInt& = 0: EXIT FUNCTION
        IF ASC(a$) = 45 THEN
                sign% = -1
                start% = 1
        ELSE
                sign% = 1
        END IF
        FOR index% = start% + 1 TO LEN(a$)
                result& = result& * 10 + ASC(MID$(a$, index%)) - 48
        NEXT
        ValInt& = result& * sign%
    END FUNCTION
```

Se si rinuncia alla capacità del comando SOUND di produrre suoni in background, cioé mentre il programma continua ad eseguire altre istruzioni, è possibile produrre il medesimo effetto manipolando direttamente le porte di I/O del microprocessore collegate all'altoparlante; notate che il listato che segue fa uso della procedura **Pause** definita in precedenza:

```
    FUNCTION DoSound (frequency%, duration%)
        '==========================================================
        ' Suona il cicalino
        ' come il comando SOUND ma utilizza esclusivamente
        ' valori interi e non suona in background
        '
        '   FREQUENCY è la frequenze del suono
        '   DURATION è la durata, espressa in ticks (1/18 di sec)
        '==========================================================
        STATIC alreadyCalled%
        DIM value&, ticks%
        ' la prima volta che questa routine è chiamata occorre
        ' inizializzare il beeper
        IF alreadyCalled% = 0 THEN
                alreadyCalled% = -1
                OUT 67, &HB6
        ENDIF
        ' imposta il valore della frequenza
        value& = 1193280 \ frequency%
        OUT 66, value& AND 255
        OUT 66, value& \ 256
        ' attiva il cicalino (bit 0 e 1 della porta 97)
        OUT 97, INP(97) OR 3
        ' attendi per il numero di ticks indicato da DURATION
        Pause duration%
        ' disattiva il cicalino
        OUT 97, INP(97) AND &HFC
    END SUB
```

Per quanto riguarda la funzione RND è possibile creare un generatore di numeri psudo-casuali interi usando una delle moltissime formule adatte allo scopo: la funzione **RndInt** usa ad esempio il metodo della congruenza lineare

```
FUNCTION RndInt%
	'=====================================================
	' Restituisce un valore intero pseudo casuale
	' nell'intervallo 0-32767
	'
	' la prima volta che viene richiamata provvede a
	' inizializzare il generatore usando il valore del
	' clock di sistema (funzione SystemTicks)
	'=====================================================
	STATIC seed%, alreadyCalled%
	IF alreadyCalled% = 0 THEN
		alreadyCalled% = -1
		seed% = (SystemTicks& AND &H7FFF)
	ENDIF
	seed% = (seed% * 35821 + 1) AND &H7FFF
	RndInt% = seed%
END FUNCTION
```

Notate che la prima volta che questa funzione viene richiamata inizializza il generatore di numeri pseudo-casuali usando il valore restituito da **SystemTicks**.

VETTORI E STRINGHE IN 'DGROUP'

Il BASIC può allocare le proprie variabili in due differenti aree di memoria: un'area detta near heap o DGROUP, e un'altra detta far heap. Tutte le variabili numeriche semplici sono memorizzate in DGROUP, mentre i vettori dinamici sono sempre memorizzati nel far heap (ricordiamo che un vettore è dinamico quando il numero dei suoi elementi non è una costante oppure quando è preceduto da una direttiva $DYNAMIC).

A differenza di altri linguaggi, il programmatore BASIC non deve necessariamente conoscere i dettagli su come e dove le variabili sono conservate in memoria per il corretto funzionamento di un programma, ma questo aspetto diventa importante quando si vuole che il programma sia il più veloce possibile. Tutti i dati nel near heap sono letti e scritti più rapidamente degli altri in quanto il registro di segmento DS del 8086 punta costantemente a quest'area di memoria, e non è necessario modificare il suo valore né eseguire un segment overloading per accedere ai dati: se non avete mai lavorato in Assembly probabilmente questi termini non vi diranno nulla, ma non importa: quel che è veramente importante è ricordarsi che per la massima velocità i dati devono essere in DGROUP. Ad esempio, il programma riportato di seguito crea un vettore dinamico di 1000 interi e lo ordina con l'algoritmo bubble sort: è facile verificare che rimuovendo la direttiva $DYNAMIC otteniamo un pro-

gramma circa tre volte più veloce ! (l'incremento di velocità è evidente solo compilando i programmi, mentre è decisamente più modesto all'interno dell'interprete)

```
REM  Ordinamento di un vettore con bubble sort
DEFINT A-Z
'$DYNAMIC
' crea il vettore e riempilo con valori casuali
DIM a(1000)
FOR i = 1 TO 1000
      a(i) = RND * 10000
NEXT
tempo! = TIMER

' ordina il vettore mediante bubble sort
FOR i = 1000 TO 1 STEP -1
      FOR j = 1 TO i - 1
              IF a(j) > a(j + 1) THEN SWAP a(j), a(j + 1)
      NEXT
NEXT
PRINT "Tempo impiegato: "; TIMER - tempo!
```

Se ci tenete a scrivere programmi veloci, evitate a tutti i costi gli array huge, ossia i vettori più grandi di 64K; infatti, se un programma è compilato con l'opzione /AH tutti gli accessi agli array avvengono non più per mezzo di lettura e scrirttura diretta della memoria, bensì tramite una chiamata ad una subroutine; per dare un'idea di quanto questo penalizzi le prestazioni del programma, si pensi che il programma precedente diventa oltre sei volte più lento se il programma è compilato con l'opzione /AH. Si può obbiettare che se il programma deve necessariamente gestire una gran mole di dati non si può proprio fare a meno degli array huge; in realtà spesso i dati si possono ristrutturare in modo da non superare quel limite. Ad esempio, il seguente frammento programma:

```
TYPE datiAnagrafici
      nome AS STRING * 20
      indirizzo AS STRING * 20
      cap AS STRING * 5
      citta AS STRING * 19
END TYPE
'$DYNAMIC
DIM rubrica(3000) AS datiAnagrafici
```

deve essere per forza compilato con /AH; possiamo però scrivere qualcosa del genere

```
'$DYNAMIC
DIM rubricaNome(3000) AS STRING * 20
DIM rubricaIndirizzo(3000) AS STRING * 20
DIM rubricaCap(3000) AS STRING * 5
DIM rubricaCitta(3000) AS STRING * 19
```

e compilare senza ricorrere agli array huge; ovviamente rinunciamo alla comodità dei record di dati strutturati, ma se la velocità è un fattore davvero importante, questa è sicuramente la scelta migliore. Un'altra tecnica molto

efficiente e quasi del tutto sconosciuta consiste nel compilare con /AH solo i moduli che devono accedere agli array huge, ed eliminare questa opzione negli altri moduli (per fare questo dovrete però compilare "manualmente" dal prompt di sistema).

In QuickBASIC 4.0 e 4.5 le stringhe a lunghezza variabile sono conservate sempre nel near heap, e quindi non c'è molto da dire. Il BASIC PDS ha introdotto le far string che, come suggerisce il loro nome, sono memorizzate all'esterno di DGROUP e pertanto la loro manipolazione è leggermente meno veloce, all'incirca del 10%, per cui se il vostro programma deve essere il più veloce possibile dovreste evitare di compilare usando l'opzione /Fs. Nel Visual BASIC per Dos non c'è la possibilità di scegliere, in quanto tutte le stringhe sono far.

Un'aspetto documentato ma poco noto del BASIC PDS è che i buffer dei file RANDOM usano lo stesso segmento usato per le stringhe, quindi DGROUP se si usano le stringhe near, oppure un altro segmento se si usano le stringhe far; di conseguenza, per quanto detto prima tutte le operazioni con i file sono più lente se il programma è compilato con l'opzione /Fs.

LA 'GARBAGE COLLECTION'

Uno dei maggiori vantaggi del BASIC su altri linguaggi è la gestione dinamica delle stringhe; ad esempio, in C o in Pascal è necessario dichiarare in anticipo la lunghezza massima di una stringa, con l'inconveniente di sprecare spesso inutilmente dei byte oppure, peggio, di scoprire nel corso dell'elaborazione che la memoria allocata allo scopo non è sufficiente. Nel BASIC invece la gestione della memoria utilizzata dalle stringhe è interamente a carico del linguaggio, e questo rende le operazioni con le stringhe altamente efficienti. Per ottenere il massimo delle prestazioni è però opportuna qualche precisazione su come il BASIC gestisce le stringhe a lunghezza variabile (attenzione quindi: le considerazioni che seguono non valgono per le stringhe a lunghezza fissa).

Iniziamo col ricordare che tutte le stringhe in QuickBASIC sono conservate in DGROUP, e lo stesso accade con i programi compilati con BASIC 7.x senza l'opzione /Fs; usando invece le far string solo le stringhe temporanee sono conservate in DGROUP, mentre le altre sono memorizzate in un segmento a parte; in ogni caso, quanto segue si applica ad entrambi i tipi di stringhe.

Ogni volta che si assegna un valore ad una variabile stringa, e tale nuovo valore differisce in lunghezza dal vecchio, il BASIC alloca per la stringa una nuova area

nel segmento appropriato (DGROUP oppure il segmento usato dalla stringa, a seconda dei casi) e marca i byte occupati dal vecchio valore come "non utilizzati", anche se questi non sono immediatamenti messi a disposizione per nuove stringhe. Dopo un certo numero di assegnazioni la memoria libera nel segmento interessato diventa insufficiente, ed in quel momento il BASIC effettua la cosiddetta garbage collection, che compatta la memoria usata dalle stringhe e "reclama" la memoria marcata come non utilizzata, in modo da creare un unico blocco di memoria libera delle maggiori dimensioni possibili. E' facile immaginare che questo processo richieda un certo tempo, ed è pertanto importante limitare la frequenza con cui esso avviene.

Vi sono numerosi metodi per ridurre il numero delle occasioni in cui il BASIC è costretto ad allocare nuova memoria per una stringa; uno dei più efficaci è di ricorrere quando è possibile alle istruzioni LSET e RSET. Il manuale del BASIC insegna che queste istruzioni sono utili per le assegnazioni alle stringhe dichiarate in una istruzione FIELD, perché esse garantiscono che la stringa non si sposti in memoria; in realtà esse funzionano perfettamente anche con le stringhe normali, e proprio perché assicurano che la lunghezza rimane invariata esse non rendono mai necessario il ricorso alle routine di allocazione/ deallocazione. Questa tecnica risulta efficace in numerose situazioni: ad esempio

```
REM restituire in padded$ il contenuto di work$, aggiungendo
REM spazi per raggiungere la lunghezza di 80 caratteri
padded$ = work$ + SPACE$(80 - LEN(work$))
```

Le istruzioni precedenti, che tra l'altro falliscono se work$ è più lunga di 80 caratteri, hanno lo svantaggio di richiedere alcuni calcoli aritmetici, e soprattutto di creare una stringa temporanea (quella creata dalla funzione SPACE$) che continua ad occupare memoria fino alla garbage collection successiva. Un risultato identico è ottenuto dalle istruzioni seguenti:

```
padded$ = SPACE$(80)
LSET padded$ = work$
```

che però evitano il ricorso a calcoli e generano meno codice compilato; inoltre, il compilatore BC 7.x e VBDOS è abbastanza intelligente da assegnare alla variabile **padded$** la zona di memoria usata dalla stringa temporanea creata da SPACE$, col risultato che al termine della assegnazione è come se tale stringa temporanea non fosse mai stata creata (riducendo quindi il ricorso alla garbage collection).

Come le istruzioni LSET e RSET, anche l'istruzione MID$ non modifica la lunghezza di una stringa, e quindi non crea "buchi" nella memoria che devono essere poi eliminati dal processo di garbage collection; si tratta quindi di un buon candidato per le nostre tecniche di ottimizzazione. Prendiamo ad esempio il seguente frammento di codice:

```
REM rendi maiuscolo il primo carattere in work$
work$ = UCASE$(LEFT$(work$, 1)) + MID$(work$, 2)
```

Per quanto funzioni correttamente, esso è circa *nove volte più lento* del sistema seguente:

```
MID$(work$, 1, 1) = UCASE$(LEFT$(work$, 1))
```

Nel tentativo di minimizzare la allocazione/deallocazione di stringhe può essere molto utile il ricorso all'istruzione SWAP. Ad esempio, la sequenza di istruzioni:

```
b$ = a$
a$ = ""
```

può essere efficacemente sostituita da:

```
b$ = ""
SWAP a$, b$
```

Nel primo caso, infatti, il BASIC deve allocare memoria per la stringa **b$**, copiare in essa i caratteri della stringa **a$** e deallocare la memoria occupata da quest'ultima variabile; usando l'istruzione SWAP, invece, il BASIC non deve fare altro che scambiare tra loro i descrittori delle due stringhe, senza allocare o deallocare memoria e senza spostare neanche un byte.

NUMERI ANZICHE' STRINGHE

Le stringhe di caratteri sono una grande comodità, ma occorre non dimenticare che il processore vede solo ed esclusivamente numeri, e che quindi i caratteri alfanumerici sono una specie di "invenzione" del linguaggio di programmazione. Nel caso del BASIC, l'uso delle stringhe passa attraverso speciali puntatori (detti *descrittori*) che sono usati per memorizzare le informazioni che riguardano la stringa stessa, e più precisamente il suo indirizzo e la sua lunghezza. Questo significa che tutti i riferimenti a stringhe richiedono almeno un accesso in più alla memoria, poiché il BASIC deve dapprima leggere il descrittore della stringa per poter accedere al suo contenuto. Per questo motivo (e per altri ancora, incluso il problema della *garbage collection* appena descritto), tutte le operazioni con le stringhe sono sempre più lente che con i numeri interi.

Da questa considerazione derivano una serie di regole pratiche che permettono di accelerare numerose operazioni. Ad esempio, il seguente frammento di programma:

```
SELECT CASE opcode$
        CASE "+": risult = n1 + n2
        CASE "-": risult = n1 - n2
        CASE "*": risult = n1 * n2
        CASE "/": risult = n1 / n2
END SELECT
```

Può essere fruttuosamente sostituito da

```
SELECT CASE ASC(opcode$)
        CASE 43: risult = n1 + n2
        CASE 45: risult = n1 - n2
        CASE 42: risult = n1 * n2
        CASE 47: risult = n1 / n2
END SELECT
```

Il tempo richiesto dalla funzione ASC è abbondantemente recuperato durante i vari confronti, che avvengono tra numeri interi (e sono quindi realizzati direttamente con istruzioni Assembly) e non tra stringhe (che invece richiedono una chiamata ad una particolare routine di libreria). Per lo stesso motivo, quando si deve testare se una stringa è nulla, la seguente istruzione

```
IF LEN(a$) = 0 THEN
```

è migliore della più ovvia

```
IF a$ = "" THEN
```

Il vantaggio di questa tecnica diventa evidente quando aumenta il numero dei confronti da effettuare; il seguente listato mostra come costruire una procedura che ricerca una stringa in un array

```
FUNCTION SearchArray% (array$(), search$)
    '======================================================
    ' Ricerca un elemento in un array di stringhe
    ' Restituisce la posizione dell'elemento, oppure -1 se
    ' la stringa SEARCH$ non è stata trovata in ARRAY$()
    '======================================================
    DIM i%, acode%
    IF LEN(search$) = 0 THEN EXIT SUB
    acode% = ASC(search$)
    FOR i% = LBOUND(array$) TO UBOUND(array$)
        IF ASC(array$(i%)) = acode% THEN
            IF array$(i%) = search$ THEN
                SearchArray% = i%: EXIT FUNCTION
            END IF
        END IF
    NEXT
    SearchArray% = -1
END FUNCTION
```

il programma confronta il codice ASCII del primo carattere della stringa cercata con quello di ciascun elemento del vettore, e solo se questo coincide procede con il confronto tra le stringhe; compilando il programma si può notare un incremento di velocità dal 30% al 50% (il valore esatto dipende ovviamente dalla distribuzione delle stringhe nel vettore e dalla frequenza dei *false match*). La stessa tecnica si può utilizzare in molte altre situazioni, ad esempio nella ricerca binaria o nell'ordinamento di un vettore di stringhe.

Tra le altre istruzioni stringa che restituiscono un numero vi è la INSTR, che infatti risulta essere utile in numerosi casi; la seguente funzione converte in decimale da una qualunque base numerica compresa tra 2 e 36, e mostra come la funzione INSTR può essere usata al posto di una struttura SELECT CASE o IF...ELSEIF per determinare il valore di ogni singola cifra dell'argomento:

```
FUNCTION ConvertToDec&(number$, baseNum%)
      '====================================================
      ' Converte in decimale da una qualunque base <=36
      '
      ' non controlla se la stringa contiene caratteri
      ' non validi, che vengono trattati come "0"
      '====================================================
      DIM digits$, i%, result&
      digits$ = "123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ"
      FOR i% = 1 TO LEN(number$)
            result& = result& * baseNum% + INSTR(digits$, UCASE$(MID$(number$, _
                  i%, 1)))
      NEXT
      ConvertToDec& = result&
END FUNCTION
```

LE PROCEDURE E IL PASSAGGIO DEI PARAMETRI

Il vecchio e glorioso GW-BASIC disponeva di una sola istruzione che, con qualche forzatura, poteva definirsi "strutturata": si tratta della GOSUB; se la procedura richiedeva dei "parametri" essi dovevano essere passati per mezzo di variabili, e lo stesso sistema doveva essere adottato se un valore doveva essere restituito al programma. Le DEFFN erano già un passo avanti notevole, in quanto permettevano di passare degli argomenti ad una procedura, e di ottenere da questa un valore. Dal punto di vista della programmazione modulare le SUB e le FUNCTION sono indiscutibilmente migliori, in quanto possono essere situate anche in un modulo differente del programma (mentre le DEFFN devono trovarsi nel medesimo modulo) e quindi favoriscono la creazione di librerie di funzioni e delle Quick Library; questa flessibilità ha però un prezzo in termini di velocità di elaborazione e dimensioni del codice compilato.

Per comprendere meglio questo tipo di considerazioni, partiamo da un costrutto GOSUB con due "argomenti", uno costante e uno variabile (per semplicità si suppone che tutte le quantità in gioco siano INTEGER):

```
arg1 = 12: arg2 = variable: GOSUB subroutine
...
subroutine:
      result = arg1 * arg2
      RETURN
```

Il compilatore trasforma la precedente riga nelle seguenti istruzioni Assembly:

```
MOV    [arg1],12       ; assegna il valore 12 a ARG1
MOV    AX,variable
MOV    arg2,AX         ; assegna il contenuto di VARIABLE a ARG2
CALL   xxxx            ; chiama la subroutine, con indirizzo a 16 bit
```

All'interno della subroutine, recuperare il valore in ARG1 comporta l'esecuzione di una semplice MOV da un indirizzo di memoria in un registro:

```
MOV     AX,[arg1]
MUL     [arg2]                ; calcola ARG1 * ARG2
MOV     result,AX             ; assegna il risultato a RESULT
RET                           ; ritorna al programma principale
```

Vediamo ora cosa avviene quando compiliamo una chiamata ad una procedura;

```
CALL myProcedure (12, variable)
```

in questo caso gli argomenti sono passati sullo stack per *reference* (il metodo usato per default dal BASIC e da altri linguaggi tra cui il Pascal) per cui il compilatore deve generare le seguenti istruzioni

```
MOV     [temp1],12            ; memorizza 12 in una variabile temporanea
MOV     AX,OFFSET temp1
PUSH    AX                    ; push l'offset della variabile temporanea
MOV     AX,OFFSET variabile
PUSH    AX                    ; push l'offset della variabile VARIABLE
CALL    xxxx:yyyy             ; chiama la procedura, conindirizzo a 32 bit
```

Notiamo subito che per richiamare una procedura occorre eseguire più istruzioni Assembly; ogni valore passato come costante richiede la creazione di una variabile fittizia e temporanea, che quindi sottrae spazio a DGROUP; inoltre, poiché le procedure devono poter essere richiamate da qualsiasi modulo del programma, è necessario che la CALL sia del tipo a 32 bit, due byte più lunga e sensibilmente più lenta della versione a 16 bit usata per compilare l'istruzione GOSUB. Vediamo ora quello che accade nella procedura chiamata per recuperare i valori sullo stack; per prima cosa la procedura deve creare un cosiddetto *stack frame*, salvando il valore corrente del registro BP e usandolo poi per ottenere i valori caricati sullo stack dal programma chiamante

```
PUSH    BP            ; salva BP
MOV     BP,SP         ; crea lo stack frame
MOV     BX,[BP+8]     ; leggi il 1° indirizzo caricato sullo stack
MOV     AX,[BX]       ; carica in AX il valore del primo argomento
MOV     BX,[BP+6]     ; leggi il 2° indirizzo caricato sullo stack
MUL     [BX]          ; esegui la moltiplicazione ARG1 * ARG2, risultato in AX
POP     BP            ; disfa lo stack frame
RET     4             ; return FAR, scarta due argomenti dall stack
```

Come si vede, sia il passaggio dei parametri che il loro recupero all'interno della procedura sono meno efficienti che nel caso del GOSUB, e questo spiega perché se la velocità del codice e la sua dimensione sono i primi fattori per importanza, si dovrebbe preferire il vecchio e bistrattato GOSUB ai più moderni SUB e FUNCTION.

Quanto detto finora si riferisce esclusivamente alle costanti e alle variabili numeriche semplici, che com'è noto sono conservate dal BASIC in DGROUP e sono pertanto accessibili con un offset a 16 bit. Come è noto, il BASIC usa il modello di memoria MEDIUM, che prevede che le variabili semplici (e gli argomenti alle procedure) siano contenute nel medesimo segmento di dati DGROUP; se il parametro da passare sullo stack si trova invece in memoria far (ad esempio, un elemento di un vettore numerico dinamico) il BASIC è costretto a copiarlo in una variabile temporanea fittizia in GROUP e passare

l'indirizzo (a 16 bit) di questa variabile alla procedura; come se non bastasse, poiché occorre tener conto della possibilità che la procedura modifichi il parametro, al ritorno dalla procedura il BASIC deve ricopiare il contenuto della variabile temporanea nell'elemento in memoria far.

Dal punto di vista del passaggio di parametri ad una procedura le stringhe convenzionali a lunghezza variabile sono gestite dal BASIC in maniera molto simile agli INTEGER; infatti quello che viene caricato sullo stack è l'indirizzo del descrittore della stringa; poiché il descrittore è conservato in memoria near (anche se si riferisce ad una stringa far) è sempre sufficiente un offset a 16 bit. Di fatto, quindi, passare una stringa convenzionale ad una procedura richiede lo stesso tempo che passare un valore intero, sia che la stringa contenga uno o diecimila caratteri. Ovviamente, al momento di operare sulla stringa la procedura dovrà calcolare la sua lunghezza e il suo indirizzo in memoria, le quali sono operazioni relativamente lente soprattutto in VBDOS e con le stringhe far del BASIC PDS .

Con le stringhe a lunghezza fissa vi sono ulteriori problemi, in quanto internamente il BASIC vede esclusivamente stringhe convenzionali a cui si accede tramite descrittore; le stringhe a lunghezza fissa sono un specie di "invenzione" introdotta con il QuickBASIC 4.0 allo scopo di ridurre l'affollamento delle stringhe in DGROUP, e in realtà esse sono molto più simili alle variabili record che alle stringhe convenzionali vere e proprie. Ecco cosa accade internamente quando un programma richiama una procedura e passa come argomento una stringa a lunghezza fissa:

```
' questo è il programma scritto da noi...
DIM myString AS STRING * 100
CALL myProcedure (myString)

' ... e questo è invece quello che compila il BASIC
DIM myString AS STRING * 100
temp$ = myString
CALL myProcedure (temp$)
myString = temp$
temp$ = ""                  ' eleimina la stringa temporanea
```

È facile vedere che il BASIC deve perdere molto tempo per copiare il contenuto della stringa nella variabile temporanea (fittizia) e viceversa, e in questo caso tale tempo è proporzionale al numero di caratteri contenuti nella stringa; se la stringa contiene 2000 caratteri occorrerà creare una stringa temporanea di pari lunghezza, senza contare i byte occupati dal codice addizionale necessario per effettuare la copia nei due sensi; la seconda copia serve nel caso in cui la procedura chiamata modifica il parametro, e anche se questo non accade mai nella procedura in questione il compilatore non può saperlo, anche perché la procedura in questione potrebbe trovarsi in un altro modulo. Questo tipo di overhead avviene anche quando si passa una stringa a lunghezza fissa ad un comando o una funzione del BASIC, ma in questo caso il BASIC è in grado di

sapere che la procedura chiamata non altererà la stringa in questione e che quindi non è necessario eseguire la seconda copia dalla variabile fittizia alla variabile stringa a lunghezza fissa.

Il passaggio di variabili TYPE è simile al passaggio di argomenti numerici; se la variabile si trova in DGROUP (è il caso di una variabile semplice oppure di un elemento di un vettore STATIC di record) il BASIC passa alla routine il suo indirizzo a 16 bit, se invece si tratta di un elemento di un array far il compilare inserisce il codice che copia il contenuto nell'elemento in una variabile temporanea fittizia in DGROUP e passa alla procedura l'indirizzo near di questa variabile (e come al solito, al termine della procedura il BASIC ricopia il contenuto di questa variabile fittizia nell'elemento dell'array in memoria far). A questo evidente spreco di tempo si deve aggiungere la memoria near usata per la variabile fittizia: se il record è lungo 1K, altrettanti caratteri in DGROUP saranno permanentemente sottratti ad altri possibili usi.

Dopo questa lunga introduzione vediamo come è possibile ridurre l'overhead imposto dal BASIC nel passaggio dei parametri; le soluzione possibili sono molteplici:

USARE GOSUB

Per quanto è stato detto in precedenza, le GOSUB restano il sistema più efficiente per richiamare porzioni di codice condiviso, poiché la CALL è del tipo infra-segmento a 16 anziché 32 bit. Se è necessario "passare" dei parametri si possono usare a tale scopo delle variabili temporanee di nome opportuno, che possono anche essere usate per "restituire" dei valori al programma chiamante: poiché evidentemente il programmatore sa quali sono i parametri che sono realmente modificati dalla procedura egli è in grado di evitare copie ed assegnazioni inutili. L'ovvio svantaggio di questo metodo è che richiede di scrivere più righe di codice del solito e che peggiora la leggibilità e manutenibilità del codice.

USARE DEF FN

I costrutti DEF FN hanno in comune con GOSUB le chiamate infra-segmento, più efficaci delle CALL inter-segmento usate per richiamare le SUB e le FUNCTION; inoltre, poiché con DEF FN i parametri sono passati per valore anziché per riferimento, il compilatore non deve mai creare codice addizionale per ricopiare il valore nella variabile originaria al termine della chiamata, anche se la variabile si trova in memoria far.

USARE VARIABILI CONDIVISE

Si può decidere di rendere disponibili alla procedura tutti o parte dei valori che essa richiede usando allo scopo opportune variabili SHARED o COMMON SHARED anziché passandoli come argomenti; in tal modo si risparmiano circa quattro bye di codice per argomento, senza contare l'overhead per ricopiare da/a una variabile in memoria far; ovviamente si può anche adottare una tecnica "mista", in cui alcuni valori sono passati attraverso una variabile globale e altri per mezzo di argomenti.

USARE UNA VARIABILE TYPE

Questa è una delle soluzioni migliori e allo stesso tempo una delle meno utilizzate, e consiste nel memorizzare tutti gli argomenti necessari in una variabile TYPE opportuna, e passare quindi un unico argomento. Ecco un esempio:

```
TYPE dataType
        gg      AS INTEGER
        mm      AS INTEGER
        aa      AS INTEGER
END TYPE
DIM giorno AS dataType
' chiama la procedura passando la data 2 Luglio 1960
giorno.gg = 2: giorno.mm = 7: giorno.aa = 1960
CALL MostraCalendario (giorno)
...
SUB MostraCalendario (var AS dataType)
```

Se la procedura deve restituire uno o più valori può usare le componenti della stessa variabile TYPE. Oltre a ridurre il numero di byte necessari a passare gli argomenti, questo metodo evita le operazioni di copia che il compilatore BASIC inserisce nel codice a nostra insaputa; purtroppo quest'ultimo vantaggio si perde se tra gli argomenti da passare alla procedura vi sono anche stringhe, che devono essere giocoforza memorizzate nel record come stringhe di lunghezza fissa: in tal caso l'overhead evitato in fase di passaggio dei parametri si ripropone all'interno della procedura, per giunta ogni volta che la stringa stessa sarà usata in qualsiasi operazione.

LA DIRETTIVA BYVAL

In QuickBASIC e BASIC PDS 7.0 la direttiva BYVAL è disponibile solo per gli argomenti passati alle routine C e Assembly, ma in BASIC PDS 7.1 e VBDOS

è anche possibile dichiarare BYVAL un argomento passato ad una procedura o funzione scritta in BASIC. Avvisando il compilatore che un parametro è passato per valore anziché per indirzzo si ottiene un duplice risparmio in velocità e compattezza del codice; facendo riferimento all'esempio visto in precedenza, si noti il risparmio in termini di istruzioni Assembly necessarie a passare una costante e una variabile INTEGER ad una procedura

```
MOV     AX,12          ; passa per valore la costante 12
PUSH    AX
PUSH    [variable]     ; passa per valore il contenuto di VARIABLE
CALL    xxxx:yyyy      ; chiama la procedura, con indirizzo a 32 bit
```

Anche all'interno della procedura i valori passati sullo stack con BYVAL sono recuperabili usando meno istruzioni

```
PUSH    BP             ; salva BP
MOV     BP,SP          ; crea lo stack frame
MOV     AX,[BP+8]      ; leggi il 1° argomento (12)
MUL     [BP+6]         ; moltiplica per il 2° argomento (VARIABLE)
POP     BP             ; disfa lo stack frame
RET     4              ; ritorna scartando due argomenti
```

Un ulteriore vantaggio dei parametri passati con BYVAL è di fornire al compilatore la preziosa informazione che la procedura non altererà l'argomento, e pertanto non sarà necessario ricopiare il valore restituito nella variabile orginaria se questa si trova in memoria far. L'istruzione BYVAL ha uno svantaggio di cui occorre essere a ben consci; i valori sono passati come una o più word a 16 bit sullo stack, per cui ad esempio un intero LONG o un valore SINGLE richiederà due operazioni di PUSH, mentre i numeri DOUBLE e CURRENCY ne richiederanno quattro; per questo motivo la direttiva BYVAL si usa in pratica quasi esclusivamente con i valori INTEGER.

ESPRESSIONI BOOLEANE E SHORT CIRCUIT EVALUATION

Il termine *short circuit evaluation* indica una tecnica utilizzata dal BASIC 7.x e VBDOS (ma non dal QuickBASIC) per evitare calcoli non necessari all'interno di una espressione booleana. L'applicazione più evidente si ha nel caso delle istruzioni IF

```
IF x > 0 AND y < 0 AND z < 0 THEN
```

se durante l'esecuzione accade che la variabile X è minore o uguale a zero, il primo confronto fallisce e rende falsa l'intera espressione: il compilatore riconosce questa situazione e correttamente fa in modo che i due test successivi non siano neanche eseguiti; la stessa cosa accade ovviamente se il primo test è soddisfatto, ma la variabile Y è maggiore o uguale a zero, nel qual caso il terzo test è evitato. Lo stesso tipo di considerazioni si possono fare per l'OR booleano:

```
IF x <> 0 OR y <> 0 THEN
```

in tal caso, se X è diverso da zero, il test su Y non è effettuato perché l'intera espressione è sicuramente vera.

Anche se questo tipo di ottimizzazioni sono introdotte automaticamente dal compilatore, il programmatore può fare il suo meglio per ottenere che esse siano sfruttate al meglio. Questo significa, ad esempio, che in una serie di test con AND è conveniente porre per primo il confronto che ha più probabilità di fallire e di risparmiare quindi il calcolo dei test successivi; analogamente, in una serie di OR booleani converrà sistemare per primo il test che ha la maggiore probabilità di risultare vero.

Una raccomandazione importante: la short circuit evaluation è attiva solo nei programmi compilati, mentre non lo è nei programmi interpretati. Di solito questo non è un problema, ma occorre stare bene attenti, in quanto è abbastanza facile introdurre alcune sottili differenze tra i programmi interpretati e quelli compilati, con la possibilità che un bug si manifesti solo a programma ultimato. Facciamo un esempio:

```
IF x = 0 AND Normalizza(y) > 0 THEN
```

Se la procedura **Normalizza** modifica il valore del suo argomento oppure di una variabile globale, nell'interprete questa modifica avverrà ad ogni esecuzione della linea, mentre nel programma compilato questa modifica avverrà solo se **X** assume un valore nullo. Può sembrare una differenza da poco, ma si tratta invece di uno dei bug più insidiosi e difficili da individuare, proprio perché i programmatori BASIC hanno la tendenza a considerare "completamente testato" un programma che sembra funzionare bene all'interno dell'ambiente di sviluppo.

Esistono altre tecniche per ottimizzare il calcolo di una espressione booleana; per comprenderle a fondo occorre tenere presente che le cosiddette operazioni booleane sono in realtà operazioni sui singoli bit degli operandi. Questo significa che l'espressione

```
IF x <> 0 OR y <> 0 THEN
```

può essere sostituita dalla più efficiente:

```
IF x OR y THEN
```

Infatti basta una qualsiasi bit non nullo in uno dei due operandi per rendere non nullo il risultato dell'operazione di OR. Non si può però fare lo stesso discorso con l'operatore AND; ad esempio, si potrebbe essere tentati di considerare equivalenti le seguenti linee di programma:

```
IF x = 0 AND y = 0 THEN
IF x AND y THEN
```

ma ciò non è corretto; per rendersene conto basta considerare un esempio numerico

```
IF 3 AND 8 THEN PRINT "Test OK"
```

ci aspetteremmo che il test dia risultato positivo e che sia stampata la stringa **"Test OK"**, ma questo non avviene. Il motivo diventa chiaro quando si considera la rappresentazione binaria dei due argomenti

```
3       = 00000000 00000011
8       = 00000000 00001000
3 AND 8 = 00000000 00000000
```

quindi l'AND fra due numeri diversi da zero può in effetti dare zero come risultato; se vogliamo possiamo però ottimizzare parzialmente l'espressione in questo modo:

```
IF 3 <> 0 AND 8 THEN PRINT "Test OK"
```

in quanto se il test è soddisfatto l'operatore booleano "<>" restituisce il valore -1, che in binario corrisponde a "11111111 11111111" e che fornisce sempre un valore non nullo quando se ne calcola il valore in AND con un qualsiasi numero in cui anche un solo bit è impostato ad uno.

Giacché siamo in argomento, alcune proprietà degli operatori booleani permettono in alcuni casi di risparmiare qualche ciclo di clock rispetto alle solite operazioni del BASIC. Ad esempio, vi sono almeno tre modi per determinare se un numero è pari

```
IF (numero% MOD 2) THEN pari% = -1 ELSE pari% = 0
IF (numero% AND 1) THEN pari% = -1 ELSE pari% = 0
pari% = (numero% AND 1) <> 0
```

ed ognuno è più efficiente del precedente; questa tecnica può essere usata in tutti i casi in cui il divisore (il secondo operando della operazione di modulo) è una potenza del due, ad esempio:

```
' calcola il resto della divisione per 1024
resto% = numero% AND 1023
```

Ecco altri usi "creativi" dell'operatore AND

```
' arrotonda verso il numero pari minore o uguale
IF (numero% AND 1) THEN numero% = numero% - 1
' questa seconda soluzione è più efficiente
numero% = numero% AND &HFFFE

' controlla che un numero sia compreso tra 256 e 1024
IF numero% >= 256 AND numero% < 1024 THEN
' stesso controllo, ma leggermente più efficiente
IF (numero% AND &H300) <= &H300 THEN
```

Anche l'operatore OR può rilevarsi utile in casi insospettati

```
' arrotonda verso il numero dispari maggiore o uguale
IF (numero% AND 1) = 0 THEN numero% = numero% + 1
' questa seconda soluzione è più efficiente
numero% = numero% OR 1
```

Un ultima cosa: molti programmatori sono abituati a testare se un numero è diverso da zero in questo modo

```
IF x THEN PRINT "X diverso da zero"
```

in luogo di

```
IF x <> 0 THEN PRINT "X diverso da zero"
```

Il sistema funziona perfettamente, ma non ci si illuda di ottenere in questo modo un programma più efficiente, in quanto il compilatore genera le stesse istruzioni Assembly in entrambi i casi, almeno nel caso in cui **X** è una variabile INTEGER, SINGLE, DOUBLE o CURRENCY. Solo nel caso in cui si sta testando una variabile di tipo LONG il primo stile può effettivamente comportare un piccolo incremento di prestazioni.

L' ISTRUZIONE 'ON ERROR'

Chiunque abbia intenzione di scrivere programmi BASIC veramente efficienti dovrebbe essere conscio dell'overhead che comporta l'istruzione ON ERROR. Infatti, quando una di queste è presente nel programma sorgente, il compilatore aggiunge automaticamente quattro byte *per ciascuna istruzione del programma*. Inutile dire che in tal modo le dimensioni dell'eseguibile aumentano notevolmente, e che l'esecuzione risulta sensibilmente più lenta.

Per quanto non possiamo modificare il comportamento del compilatore, possiamo fare parecchio per minimizzare l'impatto di questo overhead con una serie di accorgimenti:

1. uno dei sistemi più efficaci è sicuramente quello di suddividere il programma in moduli, e fare in modo che tutte le parti del programma che richiedono un gestore di errori (ad es. tutte le routine che accedono ai file) siano concentrate in un unico modulo, che sarà poi l'unico ad essere compilato con l'opzione /E o /X; quasi sempre conviene che questo sia il modulo principale del programma, poiché in tal modo ci si può "proteggere" anche dagli errori provocati in moduli secondari richiamati da quello principale
2. se si utilizza il BASIC 7.x o VBDOS, che prevedono il costrutto ON LOCAL ERROR, in alternativa o in aggiunta al metodo precedente si può pensare di costruire delle *wrapper procedure* per tutti i comandi potenzialmente in grado di generare errore, e richiamare queste nuove procedure invece dei comandi BASIC standard; il listato che segue chiarisce il concetto, creando alcune subroutine che duplicano le istruzioni OPEN e LINE INPUT# ma che restituiscono un codice di errore che può essere testato ad operazione avvenuta

```
SUB OpenFileForInput (filename$, filenum%, errcode%)
    '===========================================================
    ' Apre un file per l'input, restituendo un eventuale
    ' codice di errore in ERRCODE (richiede PDS o VBDOS)
    '
    ' se in ingresso FILENUM è nullo, viene usato il primo
    ' numero di file disponibile, ed il suo valore è restituito
    ' al programma chiamante attraverso l'argomento stesso
```

```
        '==========================================================
        ON LOCAL ERROR RESUME NEXT
        IF filenum% = 0 THEN filenum% = FREEFILE
        OPEN filename$ FOR INPUT AS filenum%
        errcode% = ERR
END SUB

SUB FileLineInput (filenum%, inputText$, errcode%)
        '==========================================================
        ' Legge una stringa da un file, restituendo un eventuale
        ' codice di errore in ERRCODE (richiede PDS o VBDOS)
        '==========================================================
        ON LOCAL ERROR RESUME NEXT
        LINE INPUT#filenum%, inputText$
        errcode% = ERR
END SUB
```

ovviamente queste *wrapper procedure* saranno riunite in un solo modulo, l'unico che sarà compilato con /X

3. meglio ancora è utilizzare, quando è possibile, delle procedure che svolgono gli stessi compiti, ma senza richiedere un gestore di errori; in altri capitoli di questo testo sono riportate tecniche alternative che prevedono l'accesso diretto al sistema operativo e che permettono di evitare del tutto l'uso della istruzione ON ERROR

OTTIMIZZARE I CICLI

La maggior parte del tempo di elaborazione viene spesa nei cicli FOR...NEXT e DO...LOOP, per cui vale la pena di ottimizzare al massimo questo tipo di strutture. Il primo consiglio a cui attenersi *sempre* è di usare variabili INTEGER per i cicli FOR...NEXT; in alcuni casi può sembrare inevitabile usare variabili floating point, ma a ben vedere esiste sempre la possibilità di guadagnare qualche frazione di secondo usando dei valori interi:

```
' mostra la tangente degli angoli tra 0 e 1 radianti
' con incrementi di 0.01
FOR angle# = 0 TO 1 STEP 0.01
        PRINT x =; angle#;    TAN(x) =; TAN(angle#)
NEXT


' versione ottimizzata
angle# = 0
FOR i% = 0 TO 100
        PRINT x =; angle#;    TAN(x) =; TAN(angle#)
        angle# = angle# + 0.01
NEXT
```

La maggiore velocità di esecuzione si spiega col fatto che nella versiona ottimizzata si evita il confronto tra la variabile **angle#** e il limite superiore del ciclo. Una tecnica "classica" di ottimizzazione consiste nella eliminazione

delle espressioni *loop invariant*, ossia di quelle espressioni che non variano all'interno del ciclo

```
FOR angle1# = 0 TO 1 STEP 0.01
        FOR angle2# = 0 TO 1 STEP 0.01
                sum# = sum# + SIN(angle1#) * SIN(angle2#) * 2
        NEXT
NEXT

' versione ottimizzata rimuovendo la loop invariant
FOR angle1# = 0 TO 1 STEP 0.01
        temp# = SIN(angle1#) * 2
        FOR angle2# = 0 TO 1 STEP 0.01
                sum# = sum# + temp# * SIN(angle2#)
        NEXT
NEXT
```

Da notare che alcuni compilatori C e C++ sono in grado di eseguire automaticamente questo tipo di ottimizzazione; noi programmatori BASIC dobbiamo invece fare da soli, ma tutto sommato non c'è da dispiacersi troppo, anche perché in questo modo si ha un maggior controllo sul codice macchina generato dal compilatore. Una variante di questa tecnica consiste nel cercare di evitare i riferimenti ad elementi di un array all'interno di un ciclo, soprattutto se si tratta di un array dinamico conservato al di fuori di DGROUP

```
' moltiplica tutti gli elementi di una matrice per un elemento
'  di un vettore
FOR col% = 1 TO numcolonne%
        FOR riga% = 1 TO numrighe%
                matrice#(riga%, col%) = matrice%(riga%, col%) * vett#(col%)
        NEXT
NEXT

' versione ottimizzata
FOR col% = 1 TO numcolonne%
        temp# = vett#(col%)
        FOR riga% = 1 TO numrighe%
                matrice#(riga%, col%) = matrice%(riga%, col%) * temp#
        NEXT
NEXT
```

È evidente che in questo modo evitiamo al BASIC la perdita di tempo associata al calcolo dell'offset dell'elemento all'interno del vettore; ecco un altro esempio

```
' moltiplicazione tra due matrici
FOR riga% = 1 TO righe%
        FOR col% = 1 TO colonne%
                FOR i% = 1 TO maxi%
                        matriceC(riga%, col%) = matrice(riga%, col%) + _
                                matriceA(riga%, i%) * matriceB(i%, col%)
                NEXT
        NEXT
NEXT

' moltiplicazione tra due matrici - versione ottimizzata
DIM matriceC(righe%, colonne%)
FOR riga% = 1 TO righe%
```

```
        FOR col% = 1 TO colonne%
                temp# = 0
                FOR i% = 1 TO maxi%
                        temp# = temp# + matriceA(riga%, i%) * matriceB(i%, col%)
                NEXT
                matriceC(riga%, col%) = temp#
        NEXT
NEXT
```

In questo caso il guadagno è sensibilmente maggiore, in quanto l'elemento rimpiezzato con la variabile temporanea **temp#** appariva due volte nel loop più interno. Un'altra tecnica spesso citata sui testi ed usata dai compilatori più evoluti è il cosiddetto *loop unrolling*, molto utile quando il corpo del ciclo contiene poche semplici istruzioni

```
' inizializza un vettore di 1000 elementi con i numeri naturali
FOR i = 1 TO 1000
        array(i) = i
NEXT

' versione ottimizzata con loop unrolling
FOR i = 1 TO 1000 STEP 2
        array(i) = i
        array(i + 1) = i + 1
NEXT
```

I guadagni ottenuti con questa tecnica non superano il dieci per cento del tempo di elaborazione complessivo, ma si tratta comunque di un espediente da tenere sempre a portata di mano quando se ne presenta la necessità. Per terminare, un consiglio semplice ma efficace: se il corpo di un ciclo DO...LOOP conta poche istruzioni inserite se possibile il controllo di uscita alla fine del ciclo anziché all'inizio; in questo modo darete la possibilità al compilatore BC.EXE di introdurre nel codice una istruzione JMP SHORT (due byte) anziché JMP (tre byte, più lenta della precedente):

```
' leggi fino a quando vi sono dati
i = 0
DO UNTIL EOF(1) OR i = UBOUND(a$)
        i = i + 1
        LINE INPUT#1, a$(i)
LOOP


' se il file contiene almeno una stringa possiamo riscrivere
' il frammento precedente in questo modo
i = 0
DO
        i = i + 1
        LINE INPUT#1, a$(i)
LOOP UNTIL EOF(1) OR i = UBOUND(a$)
```

CALCOLI IN FLOATING POINT

Tra le operazioni più lente che possono essere eseguite da un computer vi sono quelle sui numeri in virgola mobile; se alcune routine fanno uso intensivo di calcoli floating point è opportuno concentrare i nostri sforzi di ottimizzazione su di esse, poiché quasi certamente assorbono la maggior parte del tempo di elaborazione dell'intero programma.

Il primo consiglio, banale, è che se un programma di questo tipo è troppo lento dovreste prendere in considerazione l'acquisto di un coprocessore matematico; l'aumento di velocità che ne deriva è di molto superiore a quello ottenibile con qualunque tecnica di ottimizzazione software. Se avete deciso di fare a meno del coprocessore, oppure il programma è troppo lento anche sui sistemi dotati di coprocessore, non rimane che rimboccarsi le maniche e provare a spremere qualche ciclo di clock qua e là. Quelli che seguono sono alcuni consigli da usare come linee guida.

USARE COSTANTI E VARIABILI SINGLE

Se il sistema non usa il coprocessore e la precisione richiesta dai calcoli lo consente, è possibile ottenere un guadagno in velocità usando variabili SINGLE anziché DOUBLE; solitamente questo incremento non supera il 10-20%, ma se il valore esatto dipende ovviamente dal programma; un'ulteriore vantaggio di questo metodo è nel risparmio di memoria, tanto più grande se il programma deve processare vettori e matrici di grandi dimensioni. Se il computer è dotato di coprocessore in genere il guadagno è trascurabile.

Nell'impostare i calcoli in precisione singola si tenga a mente che la maggior parte delle funzioni aritmetiche del BASIC (LOG, EXP, SQR e tutte le funzioni trigonometriche) restituisce un valore SINGLE se il loro argomento è INTEGER o SINGLE, e restituisce un valore DOUBLE negli altri casi; se un valore di una espressione è in doppia precisione, potete usare la funzione CSNG per evitare che l'intera espressione sia calcolata in doppia precisione

```
result! = result! * CSNG(variable#)
```

USARE COSTANTI E VARIABILI CURRENCY

Se lavorate con BASIC PDS o VBDOS, la precisione di cinque cifre decimali è sufficiente e i calcoli da effettuare sui numeri consistono quasi esclusivamente di addizioni e sottrazioni potete anche prendere in considerazione l'idea di

usare il tipo CURRENCY. Una avvertenza importante: le moltiplicazioni e le divisioni tra valori CURRENCY sono addirittura più lente di quelle sui valori DOUBLE, per cui effettuate sempre dei benchmark per assicurarvi che la trasformazioni porti effettivamente un incremento di velocità.

USARE COSTANTI E VARIABILI LONG

Questa tecnica è abbastanza complessa e non sempre è applicabile, ma promette i migliori risultati; l'idea è abbastanza semplice: se tutte le quantità in gioco sono nello stesso ordine di grandezza è possibile effettuare uno *scaling* per portarle nel range dei numeri LONG e usufruire di nove cifre di precisione. Chiarirò il concetto con un esempio: supponiamo di dover eseguire molti calcoli su un vettore di numeri reali, tutti compresi nell'intervallo 0-10, e di accontentarci della precisione di quattro cifre decimali; ecco come affrontare il problema usando esclusivamente numeri LONG

```
DIM array&(numEls)
' i numeri nel vettore sono i valori reali moltiplicati per 10000
FOR i = 1 TO numEls
        array&(i) = arrayOfReal#(i) * 10000
NEXT
' le addizioni e le sottrazioni sono molto semplici, e non comportano
' alcun onere addizionale
' ad esempio, per sommare a tutti gli elementi il valore 1.2308
FOR i = 1 TO numEls
        array&(i) = array&(i) + 12308&
NEXT
' nelle moltiplicazioni occorre ricordarsi di "scartare" quattro cifre
' poiché nella realtà il risultato avrebbe otto cifre decimali, ma noi
' ne dobbiamo conservare solo quattro
' ad esempio, moltiplichiamo ogni elemento per il successivo
FOR i = 1 TO numEls - 1
        ' le parentesi sono molto importanti !
        array&(i) = (array&(i) * array&(i + 1)) \ 10000
NEXT
' nelle divisione ci troviamo nella situazione opposta, e dobbiamo
' moltiplicare il dividendo per 10000 prima di eseguire la divisione
' ad esempio, dividiamo ogni elemento per il precedente
FOR i = 2 TO numEls
        array&(i) = (array&(i) * 10000&) \ array&(i - 1)
NEXT
```

Chiaramente al termine del programma è necessario convertire i risultati nel formato originario

```
FOR i = 1 TO numEls
        arrayOfReal#(i) = array&(i) / 10000
NEXT
```

Come vedete non si tratta di una tecnica di ottimizzazione semplice e "indolore", in quanto costringe a modificare pesantemente il programma, ma in compenso riesce ad assicurare un guadagno nell'ordine del 300-400% o anche più.

TABELLE DI VALORI PRECALCOLATI

Si tratta di un espediente molto efficace, in cui in un certo senzo "barattiamo" della memoria in cambio di una maggiore velocità di esecuzione. Il concetto è semplice: se una funzione è richimata spesso nel corso del programma può risultare vantaggioso calcolare il suo valore nell'intervallo di validità all'inizio dell'esecuzione, e memorizzare questi valori in una tavola (vettore o matrice) per poterli consultare velocemente. Ecco un programma che calcola tutte le terne pitagoriche (a,b,c) con i primi due termini minori di 100

```
' trova le terne pitagoriche (a,b,c) con a,b <= 100
FOR a% = 1 TO 100
        FOR b% = a% TO 100
                ' la funzione CDBL evita gli overflow
                c# = SQR(CDBL(a) * a + CDBL(b) * b)
                IF c# = INT(c#) THEN PRINT a%, b%, c#
        NEXT
NEXT

' versione ottimizzata, con pre-calcolo dei quadrati
DIM quadrati#(100)
FOR a% = 1 TO 100
        quadrati#(a%) = CDBL(a%) * a%
NEXT
FOR a% = 1 TO 100
        FOR b% = a% TO 100
                ' la funzione CDBL evita gli overflow
                c# = SQR(quadrati#(a) + quadrati#(b))
                IF c# = INT(c#) THEN PRINT a%, b%, c#
        NEXT
NEXT
```

Cronometro alla mano, la versione ottimizzata è circa il 20% più veloce dell'altra, anche tenuto conto del tempo necessario a creare la tavola dei quadrati; se la funzione da tabellare è ancora più lenta i guadagni sono persino maggiori. Riuscite ad immaginare un algoritmo MOLTO più veloce del precedente senza leggere la soluzione seguente ?

```
' versione super-ottimizzata
DIM quad%(100)
REDIM radice%(100 * 100)
FOR a% = 1 TO 100
        quad%(a%) = a% * a%
        radice%(a% * a%) = a%
NEXT
FOR a% = 1 TO 100
        FOR b% = a% TO 100
                c% = radice%(quad%(a) + quad%(b))
                IF c% THEN PRINT a%, b%, c%
        NEXT
NEXT
```

Il programma precedente "precalcola" la radice quadrata dei soli quadrati perfetti, e riesce quindi ad evitare sia l'uso della funzione SQR che il confronto **c# = INT(c#)** che rallentava entrambe le precedenti versioni; cronometro alla mano la terza versione è circa 60 (sessanta!) volte più veloce della precedente, che era già parzialmente ottimizzata rispetto alla prima versione.

ELEVAMENTO A POTENZA

Per operazioni di elevamento a potenza con esponenti interi è preferibile fare a meno dell'operatore "^" del BASIC, ed eseguire manualmente le moltiplicazioni; ad esempio, le prime potenze intere di un numero dovrebbero sempre essere calcolate in questo modo

```
quadrato# = numero# * numero#
cubo# = numero# * numero# * numero#
' quarta potenza
temp# = numero# * numero#: quartaPotenza# = temp# * temp#
```

Il calcolo della quarta potenza ci fornisce l'indizio che per calcolare **X^N** dovrebbe esistere un algoritmo più efficiente della semplice concatenazione di (N-1) moltiplicazioni; in realtà si può dimostrare che è possibile calcolare il risultato con massimo 2 * $LOG_2(N)$ moltiplicazioni, dove LOG_2 indica il logaritmo in base due. Ecco una routine che può essere usata quando l'esponente è un numero intero positivo

```
FUNCTION PowerFP# (number#, exponent%)
        '______________________________
        '  elevamento a potenza, versione ottimizzata
        '     per esponenti interi e positivi
        '______________________________
        DIM expo%, temp#, result#
        IF exponent% AND 1 THEN
                result# = number#
        ELSE
                result# = 1
        END IF
        expo% = exponent% \ 2
        temp# = number# * number#

        DO WHILE expo%
                IF expo% AND 1 THEN result# = result# * temp#
                temp# = temp# * temp#
                expo% = expo% \ 2
        LOOP
        PowerFP# = result#
END FUNCTION
```

Nel peggiore dei casi la routine precedente è 5-6 volte più veloce della funzione di libreria del BASIC, e se l'elevamento a potenza è usato spesso nel programma questo singola ottimizzazione può velocizzare l'intera applicazione in maniera apprezzabile. Se poi le quantità in gioco sono solo intere e si possono rappresentare con i numeri LONG, possiamo scrivere una variante della precedente:

```
FUNCTION PowerInt& (number#, exponent%)
        '______________________________
        '  elevamento a potenza, versione ottimizzata
        '  per esponenti interi e positivi e basi intere
        '______________________________
        DIM expo%, temp&, result&
        IF exponent% AND 1 THEN
                result& = number&
        ELSE
```

```
            result& = 1
    END IF
    expo% = exponent% \ 2
    temp& = number& * number&

    DO WHILE expo%
            IF expo% AND 1 THEN result& = result& * temp&
            temp& = temp& * temp&
            expo% = expo% \ 2
    LOOP
    PowerInt& = result&
END FUNCTION
```

Ed ottenere fino a 30-40 volte la velocità del BASIC standard !

Collegato al calcolo delle potenze vi è il problema di determinare il valore di un polinomio in un insieme di punti:

```
    c(n) * x^n + c(n-1) * x ^ (n-1) + ... + c(1) * x + c(0)
```

dove ovviamente **n** è il grado del polinomio e il vettore **c()** contiene i coefficienti associati a ciascun termine; poiché tutti gli esponenti sono interi e positivi sembrerebbe che la soluzione migliore sia di usare la nostra routine **PowerFP#**, ma in realtà esiste un algoritmo ancora più efficiente - conosciuto con il nome *regola di Horner* - che riduce il calcolo a **n** addizioni e **(n+1)** moltiplicazioni.

```
FUNCTION EvaluatePoly# (grado%, coefficienti#(), x#)
    '______________________________
    ' Valuta un polinomio per un valore di X#
    '______________________________
    DIM result#, i%
    FOR i% = grado% TO 0 STEP -1
            result# = result# * x# + coefficienti#(i%)
    NEXT
    EvaluatePoly# = result#
END FUNCTION
```

Il precedente è un ottimo esempio di come nessuna tecnica di ottimizzazione possa eguagliare l'efficienza di un buon algoritmo; questo è un concetto su cui credo di non insistere mai abbastanza. Il BASIC non è un linguaggio particolarmente efficiente, soprattutto se lo si paragona alle ultime release dei compilatori ottimizzanti per C, ma ha un pregio impagabile: evitando continue ed estenuanti ricompilazione, l'ambiente di sviluppo incoraggia la sperimentazione e la ricerca di algoritmi sempre migliori per risolvere un dato problema.

8

TIPS AND TRICKS

È mia opinione personale che un buon programmatore si riconosce dalla semplicità ed eleganza con cui è in grado di risolvere i mille problemi che si incontrano nella attività quotidiana; ognuno finesce per sviluppare un bagaglio di "trucchi del mestiere", da utilizzare quando si ripresenta la necessità. In questo capitolo rivelerò alcuni dei miei personali "segreti" che - ahimè - da ora in poi non saranno più tali.

UNA FUNZIONE COMMAND$ ALTERNATIVA

Non ho mai capito perché in Microsoft qualcuno ha deciso che la funzione COMMAND$ dovesse restituire una stringa tutta in maiuscolo; in alcuni casi questa caratteristica può essere di impaccio, ad esempio se si vuole scrivere una utility che deve distinguere tra switch in maiuscolo o in minuscolo, oppure che deve ricercare in un file una stringa passata sulla riga di comando. Alla fine ho deciso di scrivere una piccola procedura che risolve il problema:

```
FUNCTION CommandTail$
        '______________________________
        ' restituisce la stringa di comando passata al programma
        '   come COMMAND$, ma non converte in maiuscolo
        '______________________________
        DIM reg AS RegType, result$, index%
        reg.ax = &H6200                  ' richiedi il PSP
        Interrupt &H21, reg, reg         ' restituito in BX
```

```
        ' la lunghezza della stringa di comando è all'offset 128 del PSP
        ' ed è seguita dai caratteri
        DEF SEG = reg.bx
        result$ = SPACE$(PEEK(128))
        FOR index% = 1 TO LEN(result$)
                MID$(result$, index%, 1) = CHR$(PEEK(128 + index%))
        NEXT
        DEF SEG
        CommandTail$ = result$
END SUB
```

Per capire il funzionamento della routine, bisogna sapere che al lancio di un programma il Dos crea in memoria un "header" di 256 byte, detto *Program Segment Prefix* (PSP); tale header contiene numerose informazioni di grande utilità, ad esempio quali file sono aperti, l'indirizzo dell'environment associato al programma, l'indirizzo a cui saltare al termine del programma stesso, e così via. Tra queste informazioni vi è, per nostra fortuna, anche la cosiddetta *CommandTail*, ossia la stringa di comando passata al programma stesso dal prompt del Dos o da un file batch; il byte all'offset 128 nel PSP contiene la lunghezza della stringa, seguita dai caratteri che la compongono.

Rispetto alla funzione COMMAND$, la funzione **CommandTail$** presenta due differenze: (1) gli spazi e i caratteri di tabulazione iniziali non sono rimossi e (2) nei programmi interpretati restituisce la stringa di comando inviata all'ambiente di sviluppo, e non quello impostato con il comando *Run-Modify* COMMAND$. Il primo problema si risolve con una semplice funzione LTRIM$, ma in alcuni casi questo comportamento può essere utile; per risolvere il secondo problema si può scrivere qualcosa del genere

```
' memorizza la stringa di comando nella variabile CMD$
' nei programmi interpretati CMD$ conterrà quanto inserito
' mediante il comando Run-Modify COMMAND$
cmd$ = CommandTail$
IF UCASE$(LTRIM$(cmd$)) <> COMMAND$ THEN cmd$ = COMMAND$
```

Notate che la funzione **CommandTail$** può anche essere usata per distinguere se un programma BASIC è eseguito in modo interpretato o compilato, in questo modo

```
IF UCASE$(LTRIM$(CommandTail$)) <> COMMAND$ THEN intrepretato = -1
```

ESISTE UN FILE O UNA DIRECTORY ?

In QuickBASIC il modo più efficiente per controllare se un file esiste è abbastanza goffo; occorre infatti impostare un gestore di errori, poi tentare di aprire il file in modo INPUT, per poi chiudere il file nel caso l'operazione non ha prodotto errore:

```
        ON ERROR GOTO ErrorHandler
        filenum% = FREEFILE
        fileExist% = -1
        OPEN filename$ FOR INPUT AS #filenum
        IF fileExist% THEN CLOSE #filenum
        .....
    ErrorHandler:
        fileExist% = 0: RESUME
```

Gli utenti del BASIC PDS sono più fortunati, in quanto possono fare uso della funzione DIR$, anche se usata in modo poco ortodosso:

```
fileExist% = (DIR$(filename$) <> "")
```

che può anche scriversi come

```
fileExist% = (LEN(DIR$(filename$)) > 0)
```

per risparmiare qualche byte e qualche ciclo di clock. E' meno ovvio, tuttavia, come controllare l'esistenza di una directory:

```
directoryExist% = (LEN(DIR$(dirname$ + "\NUL")) > 0)
```

S'impone una spiegazione: il device driver NUL, come tutti gli altri device driver come CON, LPT1 o COM1, "appaiono" in qualunque directory li si cerchi, a condizione ovviamente che il percorso indicato esista. Usando la stessa tecnica un file batch può testare l'esistenza di una directory in questo modo:

```
IF EXIST C:\TESTDIR\NUL ECHO La directory C:\TESTDIR esiste
```

SEDICI COLORI PER IL BACKGROUND

Qualche anno apparirono i primi programmi che usavano ben 16 colori per lo sfondo in modalità testo - mi ricordo ad esempio la versione 5 di PC*Tools* - e decisi che anche i miei programmi sarebbero stati in grado di offrire le stesse prestazioni, per cui iniziai a sfogliare i vari manuali delle schede video. Alla fine scoprii che, sulle schede CGA, EGA, MCGA e VGA è possibile rinunciare al lampeggio per ottenere sedici colori anziché i soliti otto. Diamo una occhiata a come i colori sono codificati nel byte di attributi associato a ciascun carattere in modo testo:

```
bit        7      6      5      4      3      2      1      0
           flash                - sfondo -           — testo —
```

dopo il "trattamento" il significato del byte di attributi diventa invece

```
bit        7      6      5      4      3      2      1      0
           — sfondo —           — testo —
```

e appunto i quattro bit a disposizione del colore di sfondo permettono di esprimere 16 combinazioni differenti.

Il problema collegato alla attivazione di questa caratteristica è che il metodo da seguire dipende dalla scheda video: nelle più recenti EGA, MCGA e VGA si attiva con una semplice chiamata al BIOS, mentre con le vecchie CGA occorre manipolare direttamente i registri della scheda. La procedura Background16 esegue i controlli necessari, e può essere usata "a scatola chiusa" nei propri programmi

```
SUB Background16 (status%)
        '____________________________
        ' attiva/disattiva i sedici colori per lo sfondo
        '____________________________
        DIM reg AS RegType, streg%
        ' prima di tutto occorre determinare se stiamo lavorando con una
        ' scheda EGA/MCGA/VGA; questo si ottiene invocando un servizio del
        ' BIOS video che è assente nelle vecchie schede CGA, MDA, ecc.
        reg.ax = &H1200  ' richiedi la configurazione
        reg.bx = &H10
        Interrupt &H10, reg, reg
        DEF SEG = 0              ' per accedere alla memoria bassa

        IF reg.bx <> &H10 THEN
                ' se BL è stato modificato si tratta di una EGA/MCGA/VGA
                ' e possiamo richiamare un particolare servizio del BIOS
                reg.bx = -(status% = 0)  ' deve essere BX = 0 o 1
                reg.ax = &H1003              ' attiva/disattiva l'alta
                Interrupt &H10, reg, reg ' luminosità per lo sfondo
                ' altrimenti dobbiamo verificare che si tratti di una CGA e non
                ' di una scheda monocromatica MDA o Hercules
        ELSEIF PEEK(&H463) = &HD4 THEN
                streg% = PEEK(&H465)             ' ottieni il valore dello status
        ' imposta o azzera il bit 5
                streg% = (streg% AND &HFA) OR (status% = 0) AND 32
                POKE &H463, streg%               ' memorizza in memoria bassa
                OUT &H3D8, streg%                ' e nei reigstri della scheda
        END IF
        DEF SEG                                  ' ripristina DEF SEG
END SUB
```

Per attivare i sedici colori di sfondo basta allora eseguire una istruzione

```
CALL Background16 (-1)
```

ricordandosi però di disabilitare questa caratteristica prima di terminare l'esecuzione del programma, con

```
CALL Background16 (0)
```

in caso contrario tutti i programmi eseguiti in seguito potrebbero non funzionare a dovere. Dopo aver attivato i sedici colori bisogna ovviamente sapere come sfruttare questa caratteristica; se si tiene a mente che il bit addizionale a disposizione per l'attributo di colore era quella originariamente destinato al lampeggio, è abbastanza facile capire come fare uso della istruzione COLOR. Ad esempio, una scritta bianca (codice di colore pari a 7) su sfondo blu chiaro (codice 9) si ottiene con

```
COLOR 7 + 16, 1
```

dove il fattore additivo 16 indica appunto di attivare l'alta luminosità per lo sfondo; se il procedimento vi sembra macchinoso, potete inserire la seguente procedura nel programma e specificare direttamente i colori per i caratteri e per lo sfondo:

```
SUB Color2 (testo%, sfondo%)
        COLOR testo% - 16 * (sfondo% >= 8), sfondo% AND 7
END SUB
```

HARD-COPY DELLO SCHERMO

Volete lasciare un ricordo su carta del vostro programma ? Niente di meglio di un hardcopy del contenuto dello schermo. Per simulare da programma la pressione del tasto PrintScreen è sufficiente invocare l'interrupt 5

```
SUB ScreenHardCopy
        DIM reg AS RegType
        Interrupt 5, reg, reg
END SUB
```

Questo sistema funziona anche con le schermate grafiche, a condizione che sia stato preventivamente caricato in memoria il programma GRAPHICS.COM. Se invece ci possiamo accontentare dei soli modi testo, il seguente listato mostra una routine di hardcopy "selettiva", che permette di indicare l'area rettangolare dello schermo che sarà effettivamente stampata:

```
SUB WindowHardCopy (topRow%, leftCol%, bottomRow%, rightCol%)
        '______________________________
        '  hardcopy di un'area rettangolare dello schermo
        '______________________________
        DIM row%, col%
        FOR row% = topRow% TO bottomRow%
                FOR col% = leftCol% TO rightCol%
                        LPRINT CHR$(SCREEN(row%, col%));
                NEXT
                LPRINT
        NEXT
END SUB
```

STAMPARE SU LPT2 E LPT3

E' noto che l'istruzione LPRINT del BASIC invia i dati alla stampante parallela collegata alla porta LPT1; come possiamo allora approfittare di una seconda (o una terza) stampante collegata al sistema ? In BASIC "puro", senza trucchi di alcun genere, l'unica soluzione è di aprire un file e di inviare i dati usando l'istruzione PRINT#

```
lpt2% = FREEFILE
OPEN "LPT2" FOR OUTPUT AS #lpt2%
```

```
PRINT#lpt2%, messaggio$
...
CLOSE #lpt2%
```

Il sistema funziona, ma ha lo svantaggio di "rubare" un handle di file alla applicazione, e di richiedere la sostituzione di tutte le istruzioni LPRINT del programma. Lo stesso effetto, con minor sforzo, si ottiene modificando un paio di valori in bassa memoria che conservano gli indirizzi dei gestori delle porte LPT1-LPT3

```
SUB SwapLptPorts (lptnum%)
        '______________________________
        ' scambio logico di una stampante con quella collegata a LPT1
        '______________________________
        DIM loByte%, hiByte%
        ' assicursi che l'argomento sia nell'intervallo consentito
        IF lptnum% < 1 OR lptnum% > 4 THEN EXIT SUB
        ' salva l'indirizzo corrente del gestore di LPT1
        DEF SEG = 0
        loByte% = PEEK(&H408)
        hiByte% = PEEK(&H409)
        ' memorizza l'indirizzo del gestore lptnum in LPT1
        POKE &H408, PEEK(&H406 + lptnum% * 2)
        POKE &H409, PEEK(&H407 + lptnum% * 2)
        ' infine, completa lo scambio degli indirizzi
        POKE &H406 + lptnum% * 2, loByte%
        POKE &H407 + lptnum% * 2, hiByte%
        DEF SEG
END SUB
```

SwapLptPort permette di scambiare "logicamente" una qualsiasi stampante parallela con quella collegata a LPT1; eseguendo ad esempio

```
SwapLptPort 2
```

tutte le istruzioni LPRINT agiranno su LPT2 anziché LPT1; in questo caso è importante riportare la situazione alla normalità eseguendo un nuovo scambio prima del termine del programma; la cosa interessante è che questa istruzione funziona anche con i programmi esterni al BASIC e lanciati mediante SHELL, e addirittura con gli spooler di stampa come PRINT.COM, a condizione che lo swap delle porte sia eseguito prima di attivare lo spooler.

Una alternativa è quella di fare a meno sia di LPRINT che di PRINT#, e accedere alla stampante direttamente attraverso l'interrupt 17 esadecimale del BIOS, che permette di indicare a quale stampante deve essere inviato un carattere (vedere il listato 4). L'accesso diretto al BIOS ha anche un altro vantaggio rispetto alla istruzione LPRINT, in quanto gli errori possono essere trattati senza il ricorso ad un comando ON ERROR. Naturalmente, in un programma reale dovreste trattare gli errori in modo opportuno, a differenza della procedura PrintLpt che termina brutalmente mostrando un messaggio di errore.

```
SUB PrintLpt (lptnum%, message$)
        '______________________________
        ' invia una stringa ad una qualsiasi stampante parallela
        '______________________________
        DIM reg AS RegType, index%
```

```
        FOR index% = 1 TO LEN(Message$)
                ' invia ciascun carattere alla stampante - il BIOS attende
                ' il codice ascii in AL e il numero della funzione (zero) in AH
                ' il registro DL deve contenere il numero della stampante
                reg.ax = ASC(MID$(message$, index%))
                reg.dx = lptnum%
                Interrupt &H17, reg, reg
                ' uscendo dall'interrupt AH conserva il codice di errore,
                ' codificato nei bit del registro
                IF (reg.ax AND &HFF00) <> &H1000 THEN
                        IF reg.ax AND &H2000 THEN
                                PRINT "Out of paper": EXIT FOR
                        ELSE
                                PRINT "Device fault": EXIT FOR
                        END IF
                END IF
        NEXT
END SUB
```

IL TIMEOUT DELLA STAMPANTE

C'è un aspetto delle stampanti non molto noto: quando provate ad emettere un comando di stampa e non esiste alcuna stampante collegata al sistema, il messaggio di errore non appare immediatamente, ma soltanto dopo un considerevole lasso di tempo, fino a mezzo minuto. Questo è un "retaggio" delle vecchie stampanti che non rispondevano immediatamente ai segnali sulla porta parallela, ad esempio perché erano impegnate a stampare una riga di caratteri, per cui il BIOS doveva necessariamente aspettare un certo numero di secondi prima di stabilire che la mancata risposta fosse un segnale di stampante assente o mal collegata, e non più semplicemente di una stampante particolarmente lenta. Il tempo concesso dal BIOS alla stampante è detto timeout, ed è impostato al bootstrap ad un valore che va da una decina di secondi fino ad un paio di minuti (il valore esatto dipende dal particolare BIOS); tale valore è conservato in una locazione in bassa memoria, anzi in quattro distinte locazioni, una per ciascuna possibile stampante parallela.

Dati i progressi nella tecnologia, il timeout di default è troppo alto per le stampanti più recenti, per cui può essere molto conveniente diminuire il timout ad un valore più ragionevole, diciamo 3 o 5 secondi. La procedura **SetPrinterTimeout** riportata fa proprio questo. Ricordatevi che anche in questo caso l'impostazione riguarda l'intero sistema, e che il suo effetto dura anche dopo il termine dell'esecuzione del programma BASIC

```
SUB SetPrinterTimeout (lptPort%, seconds%)
        '-------------------------------------
        ' Modifica il timeout di una stampante parallela
        '-------------------------------------
        IF lptPort% < 1 OR lptPort% > 4 THEN EXIT SUB
        DEF SEG = 0
```

```
        POKE &H477 + lptPort%, seconds%
        DEF SEG
END SUB
```

Una alternativa al sistema appena illustrato consiste nel cambiare il timout solo temporaneamente, giusto per controllare che la stampante è collegata, accesa, on-line e pronta ad accettare dei dati. La funzione **IsPrinterReady**, che restituisce il valore -1 se la stampante specificata è pronta, zero altrimenti;

```
FUNCTION IsPrinterReady% (lptPort%, timeout%)
        '___________________________
        ' controlla che una stampante parallela
        ' sia pronta ad accettare dati
        '___________________________
        DIM reg AS RegType, oldTimeout%
        ' restituisci zero se lptport è fuori intervallo
        IsPrinterReady% = 0
        IF lptPort% < 1 OR lptPort% > 4 THEN EXIT SUB
        DEF SEG = 0
        ' salva il timeout corrente, imposta il nuovo timeout
        oldTimeout% = PEEK(&H477 + lptPort%)
        POKE &H477 + lptPort%, timeout%
        ' invia uno spazio, usando l'interrupt BIOS
        reg.ax = &H20
        reg.dx = lptnum%
        Interrupt &H17, reg, reg
        ' lo status è restituito in AH, di cui però ci interessano
        ' solo i bit di timeout, selected e I/O error
        reg.ax = reg.ax AND &H1900
        ' se il bit di printer selected bit è attivato possiamo
        ' continuare con il test
        IF reg.ax = &H1000 THEN
                ' invia un backspace, per "disfare" lo spazio precedente
                reg.ax = &H8
                Interrupt &H17, reg, reg
                ' controlla nuovamente lo status restituito in AH
                reg.ax = reg.ax AND &H1900
                reg.dx = lptnum%
                IF reg.ax = &H1000 THEN
                        ' se tutti i test hanno avuto successo la stampante è OK
                        IsPrinterReady = -1
                END IF
        END IF
        ' ripristina il timeout originale
        POKE &H477 + lptPort%, oldTimeout%
        DEF SEG
END FUNCTION
```

è allora possibile scrivere un frammento di codice come il seguente

```
DO UNTIL IsPrinterReady (1, 3)
        PRINT "Accendi e poni on-line la stampante LPT1"
        DO: LOOP UNTIL INKEY$ <> ""
LOOP
```

ALTRI TRUCCHI CON LA STAMPANTE

Ancora un paio di "trucchi" per la stampante, semplici ma non del tutto ovvii. Per vivacizzare i nostri report su carta spesso vorremmo inserire delle varianti sullo stile, ad esempio il grassetto, il corsivo e il sottolineato. Purtroppo non tutte le stampanti supportano i medesimi comandi, per cui occorrerebbe ogni volta studiare il manuale della particolare stampante e modificare il programma di conseguenza; se poi il programma deve girare con un gran numero di configurazioni siamo in un bel guaio, e dobbiamo scegliere tra scrivere i driver per le differenti stampanti oppure rassegnarci ad evitare del tutto orpelli vari.

Ma è proprio vero? In effetti qualche cosa si può ottenere anche senza impazzire con le migliaia di stampanti differenti sul mercato, semplicemente approfittando del fatto che praticamente tutte le stampanti supportano il carattere CHR$(8) come BACKSPACE, che sposta il cursore di un carattere di una posizione a sinistra. Ecco allora come ottenere il sottolineato con *qualsiasi* stampante:

```
LPRINT messaggio$;                         ' stampa il messaggio
LPRINT STRING$(LEN(messaggio$, 8));        ' sposta il cursore indietro
LPRINT STRING$(LEN(messaggio$;"_"));       ' aggiungi la sottolineatura
```

Se ci accontentiamo di un "doppio ribattuto" in luogo di un vero e proprio grassetto, possiamo evidenziare un messaggio usando una variante del metodo precedente

```
LPRINT messaggio$;                         ' prima battuta
LPRINT STRING$(LEN(messaggio$, 8));
LPRINT messaggio$;                         ' seconda battuta
```

OPERAZIONI CON GLI ARRAY DI BIT

In BASIC non esistono i tipi Boolean, per cui è abitudine diffusa memorizzare questi valori in una variabile INTEGER a 16 bit. Questa limitazione deventa un po' seccante quando dobbiamo trattare non con un singolo valore, ma piuttosto con qualche migliaio. Ad esempio, dovendo memorizzare 8192 valori booleani occorrerebbe un array di altrettanti elementi, con un consumo di ben 16K, laddove in teoria sarebbe sufficiente un solo Kbyte.

Per ridurre questo spreco è possibile comprimere 16 valori booleani in ciascun elemento di un array di interi. Per prima cosa, quindi, occorre dimensionare un array di dimensioni opportune: se intendiamo memorizzare **maxEls%** elementi booleani avremo bisogno di creare un array in questo modo:

```
DIM array%(0 TO (maxEls - 1) \ 16)
```

Fatto questo, possiamo usare l'array come argomento per le routine riportate di seguito, che permettono di impostare o recuperare un valore booleano da un siffatto array di bit.

```
SUB BitArraySet (array%(), index%, value%)
        '___________________________
        ' Imposta un valore booleano in un array di bit
        '___________________________
        DIM n%, mask%
        n% = index% \ 16
        mask% = Power2(index% AND 15)
        array%(n) = array%(n) AND NOT mask% OR (mask% AND value% <> 0)
END SUB


FUNCTION BitArrayGet% (array%(), index%)
        '___________________________
        ' Restituisce un valore booleano in un array di bit
        '___________________________
        BitArrayGet% = (array%(index% \ 16) AND Power2(index% AND 15)) <> 0
END FUNCTION


FUNCTION Power2% (exponent%)
        '___________________________
        '  Calcola il valore di una potenza del due
        ' (usata dalle routine BitArraySet / BitArrayGet )
        '___________________________
        SELECT CASE exponent%
                CASE 0: Power2% = 1
                CASE 1: Power2% = 2
                CASE 2: Power2% = 4
                CASE 3: Power2% = 8
                CASE 4: Power2% = &H10
                CASE 5: Power2% = &H20
                CASE 6: Power2% = &H40
                CASE 7: Power2% = &H80
                CASE 8: Power2% = &H100
                CASE 9: Power2% = &H200
                CASE 10: Power2% = &H400
                CASE 11: Power2% = &H800
                CASE 12: Power2% = &H1000
                CASE 13: Power2% = &H2000
                CASE 14: Power2% = &H4000
                CASE 15: Power2% = &H8000
        END SELECT
END FUNCTION
```

Si noti che qualunque valore non nullo per il parametro **value%** nella procedura **BitArraySet** imposta il bit ad 1, mentre la funzione **BitArrayGet** restituisce il valore 0 o -1. Entrambe le procedure fanno riferimento alla funzione **Power2**, che calcola il valore di una potenza del due ed è infinitamente più veloce dell'operazione (2^index).

RICERCA VELOCE IN UN ARRAY DI STRINGHE

Non riesco quasi ad immaginare un programma di una certa dimensione che, in un modo o nell'altro, non abbia bisogno ad un certo punto di ricercare una stringa in un vettore di stringhe. La procedura **SearchStr** risolve il problema nel modo più ovvio e scontato:

```
FUNCTION SearchStr% (array$(), search$)
        '________________________
        ' Ricerca un valore in un vettore di stringhe
        '       restituisce l'indice dell'elemento che contiene
        '       la stringa ricercata, oppure -1 se la ricerca fallisce
        '________________________
        DIM index%
        SearchStr% = -1
        FOR index% = LBOUND(array$) TO UBOUND(array$)
                IF array$(index%) = search$ THEN
                        SearchStr% = index%: EXIT FOR
                END IF
        NEXT
END FUNCTION
```

I confronti tra stringhe sono una delle operazioni più lente del BASIC, e il programma sarebbe sensibilmente più veloce se confrontasse direttamente i codici ASCII dei caratteri. La seconda versione della routine fa proprio questo, confrontando dapprima il codice ASCII del primo carattere della stringa di ricerca con il codice ASCII del primo carattere di ciascun elemento dell'array, proseguendo con il confronto delle intere stringhe solo se il primo test fornisce risultato positivo:

```
FUNCTION SearchStr2% (array$(), search$)
        '________________________
        '  Ricerca un valore in un vettore di stringhe
        '               - versione ottimizzata -
        '
        '       restituisce l'indice dell'elemento che contiene
        '       la stringa ricercata, oppure -1 se la ricerca fallisce
        '       avviene errore se la stringa di ricerca è una stringa
        '       nulla oppure se l'array contiene stringhe nulle
        '________________________
        DIM index%, acode%
        SearchStr2% = -1
        acode% = ASC(search$)
        FOR index% = LBOUND(array$) TO UBOUND(array$)
                IF ASC(array$(index%)) = acode% THEN
                        IF array$(index) = search$ THEN
                                SearchStr2% = index%: EXIT FOR
                        END IF
                END IF
        NEXT
END FUNCTION
```

Se compiliamo il programma noteremo un incremento di velocità nell'ordine del 40% (il valore esatto dipende ovviamente dalla distribuzione dei valori nell'array e dalla frequenza dei match parziali, ossia dei casi in cui i primi

caratteri coincidono), che non è affatto poco se pensiamo che si tratta di una tecnica molto semplice e di facile applicazione. Viceversa, l'incremento delle prestazioni all'interno dell'ambiente di sviluppo è quasi nullo. Si noti che, poiché la funzione ASC provoca errore se applicata ad una stringa nulla, possiamo usare questo sistema solo se siamo sicuri che l'array non contenga stringhe nulle.

CONTROLLO DELLA RETE LOCALE

Capita spesso di voler controllare se il programma sta funzionando sotto rete locale, ad esempio per attivare o impedire particolari funzioni; in altri casi può essere conveniente semplicemente avvisare l'operatore che il sistema non è collegato alla rete, e che dovrebbe uscire dal programma per attivare la rete, e così via. Per fortuna è molto semplice stabilire se il modulo SHARE.EXE (o un programma equivalente) è stato preventivamente caricato in memoria, interrogando l'interrupt 2FH; quando viene installato, infatti, SHARE si "aggancia" a questo interrupt, in modo da evitare installazioni multiple. La funzione **IsShareLoaded** riportata nel listato 7 restituisce il valore -1 se SHARE è installato, zero altrimenti:

```
FUNCTION IsShareLoaded%
        '______________________
        ' testa la presenza del modulo SHARE.EXE
        '______________________
        DIM reg AS RegType
        reg.ax = &H1000
        Interrupt &H2F, reg, reg ' interroga il multiplex int
        IF reg.flags AND 1 THEN  ' errore se il flag di carry è ON
                IsShareLoaded% = 1       ' (Dos 3.1 o precedente)
        ELSEIF reg.ax AND &HFF THEN      ' altrimenti testa AL
                IsShareLoaded% = -1      ' se <>0 SHARE è installato
        ELSE
                IsShareLoaded% = 0
        END IF
END FUNCTION
```

poiché la funzione dell'interrupt 2FH che serve a testare la presenza del modulo di rete è presente solo a partire dalla versione 3.2 del Dos, come caso speciale la funzione restituisce il valore 1 se eseguita con un Dos anteriore a questa versione.

CANCELLAZIONE FILE A PROVA DI UNDELETE

Se intendiamo cancellare un file contenente informazioni riservate, e vogliamo essere sicuri che nessuno lo "resusciti" con un comando di UNDELETE

possiamo seguire il metodo indicato dalla routine **WipeFile**, che consiste nell'aprire il file in modo BINARY e riempirlo con zeri, poi chiudere il file ed infine cancellarlo, certi che qualora il contenuto del file fosse recuperato, il curiosone di turno non troverebbe che una lunga sequenza di zeri.

```
SUB WipeFile (filename$)
        '______________________________
        '  Cancellazione file a prova di UNDELETE
        '______________________________
        DIM zeroes$, i%, filenum%
        filenum% = FREEFILE
        zeroes$ = STRING$(512, 0)
        OPEN filename$ FOR BINARY AS #filenum
        FOR i% = 0 TO LOF(1) \ 512
                PUT #filenum,,zeroes$
        NEXT
        CLOSE #filenum
        KILL filename$
END SUB
```

A differenza di quel che si potrebbe credere, il sistema funziona solo se il file si apre in modo BINARY o RANDOM e non, ad esempio, in modo OUTPUT; il motivo è che in quest'ultimo caso la lunghezza del file è azzerata dal comando OPEN, per cui i nuovi dati (i caratteri nulli) sarebbero in realtà scritti in un'altra zona del disco rigido, senza andare a sovrapporsi ai dati originali.

WINDOWS O NON WINDOWS ?

La maggior parte dei programmi scritti in BASIC girano in una finestra Dos sotto Windows senza alcun problema, ma se avete scritto un programma che per qualche motivo particolare non deve assolutamente essere eseguito in Windows, oppure deve disabilitare alcune sue funzioni all'interno di questo ambiente, troverete utile questa funzione:

```
FUNCTION WindowsMode%
        '______________________________
        ' Determina se il programma è eseguito sotto Windows
        '   restituisce  0 = eseguito in Dos
        '                1 = eseguito in Windows, modo reale o standard
        '                2 = eseguito in Windows, modo enhanced
        '______________________________
        DIM reg AS RegType
        reg.ax = &H4680                 ' testa Windows in modo reale
        Interrupt &H2F, reg, reg        ' o standard
        IF reg.ax = 0 THEN
                WindowsMode = 1         ' trovato
        ELSE
                reg.ax = &H1600
                Interrupt &H2F, reg, reg
                IF reg.ax AND &HFF THEN ' controlla AL
```

```
                    WindowsMode = 2          ' trovato il modo enhanced
                ELSE
                    WindowsMode = 0          ' Windows non trovato
                END IF
        END IF
END FUNCTION
```

NON DIMENTICATE I MONITOR MONOCROMATICI

Come potete essere certi che il vostro programma, che fa ottima mostra di se su uno schermo a colori, continui ad apparire attraente e funzionale anche su un monitor monocromatico, ad esempio su un portatile ? La soluzione più ovvia è di comprare un monitor in bianco/nero, ma certamente non è la più economica. Per fortuna esiste un rimedio pratico più intelligente. Quasi tutti gli adattatori VGA supportano un modo operativo detto *grey scale summing*, che se abilitato fornisce una approssimazione abbastanza precisa di come il vostro programma apparirà su un monitor monocromatico. Ecco una breve routine BASIC che potete includere nel vostro programma:

```
SUB SetGreyScaleSumming (status%)
        '______________________________
        ' Attiva e disattiva il grey scalu summing
        '______________________________
        DIM reg AS RegType
        ' dobbiamo chiamare la funzione 12h, sottofunzione 33h
        ' passando in AL il valore 0 per abilitare, 1 per disabilitare
        reg.ax = &H1200 - (status% = 0)
        reg.bx = &H33
        Interrupt &H10, reg, reg
        ' il grey scale summing diventa attivo solo dopo aver attivato
        ' un nuovo modo video, quindi occorre richiedere il modo corrente...
        reg.ax = &HF00
        Interrupt &H10, reg, reg
        ' ... e attivarlo nuovamente; impostando ad uno il settimo bit
        ' in AL si ottiene di NON pulire il contenuto dello schermo
        reg.ax = (reg.ax AND &HFF) OR &H80
        Interrupt &H10, reg, reg
END SUB
```

Per testare il vostro programma in modo monocromatico inserite questa istruzione all'inizio del file sorgente

```
CALL SetGreyScaleSumming (1)
```

ma ricordatevi di disabilitare il grey scale summing alla fine della applicazione, con il comando

```
CALL SetGreyScaleSumming (0)
```

altrimenti sarà necessario attendere il reset successivo.

DETERMINARE IL TIPO DI FLOPPY DISK

Determinare il numero dei floppy disk montati sul sistema, ed il tipo di ciascuno di essi, sembra essere più un esercizio di programazione che una tecnica di qualche utilità, ma sicuramente questo genere di conoscenza può rendere il vostro programma più amichevole nei confronti dell'utente. Per esempio, nelle mie applicazioni gestionali sono solito inserire un sottomenù che permette di eseguire delle azioni come la formattazione e la duplicazione di un dischetto, ma prima di eseguire il comando Dos più appropriato ho necessità sapere quanti floppy sono disponibili. Questo valore può essere calcolato abbastanza facilmente usando l'interrupt 11h del BIOS ed interpretando il risultato restituito in AX

```
FUNCTION FloppyDisks%
        '__________________________
        ' Dertermina il numero di unità floppy installate
        '__________________________
        DIM reg as RegType
        Interrupt &H11, reg, reg
        IF (reg.ax AND 1) = 0 THEN
                ' se il bit 0 di AL è zero non vi sono floppy
                FloppyDisks% = 0
        ELSE
                ' altrimenti il valore è memorizzato nei bit 6-7
                FloppyDisks% = (reg.ax AND &HC0) \ 64 + 1
        END IF
END FUNCTION
```

Determinare invece il *tipo* di ciascun drive è leggermente più difficile, in quanto è necessario accedere alle informazioni memorizzate nella memoria CMOS (ovviamente, purché il sistema disponga di memoria CMOS):

```
SUB FloppyDiskType (driveA%, driveB%)
        '__________________________
        ' Determina il tipo dei floppy A: e B:
        '__________________________
        DIM byte%
        OUT 112,16                      ' vogliamo leggere la porta 16
        byte% = 0                               ' una breve pausa
        byte% = INP(113)          ' ora possiamo leggere il valore
        ' le informazioni sul drive A: sono nei bit 4-7
        driveA% = (byte% AND &HF0) \ 16

        ' le informazioni sul drive B: sono nei bit 0-3
        driveB% = byte% AND &HF
END SUB
```

In uscita dalla procedura i parametri **driveA** e **driveB** possono assumere i seguenti valori:

0	floppy non trovato
1	360K 5.25"
2	1.2M 5.25"

3 720 3.5"

4 1.44M 3.5"

5 2.88M 3.5"

L'ETICHETTA DI VOLUME E IL NUMERO SERIALE DEL DISCO

Nella versione 4 di Ms-Dos è stato introdotto il numero seriale di disco, un numero pseudo-casuale a 32 bit creato durante la formattazione e usato probabilmente dal Dos per i suoi scopi; questo valore potrebbe però essere usato con profitto dai programmi applicativi, ad esempio nelle procedure di installazione. Sfortunatamente nei manuali ufficiali Microsoft non vi è traccia di un servizio Dos per leggere questo valore, ma se sfogliamo qualche testo alternativo troviamo che in effetti la funzione non documentata 69h serve proprio a leggere il *serial id* e l'etichetta di volume. Ecco una procedura BASIC che svolge tutto il lavoro necessario:

```
SUB GetDiskLabel (drive$, label$, serial$)
        '_______________________________
        ' Legge l'etichetta e il numero seriale di un disco
        '_______________________________
        DIM reg AS RegTypeX, buffer AS STRING * 25, id&
        IF LEN(drive$) = 0 THEN EXIT SUB
        ' questo servizio esiste solo a partire dal Dos 4
        ' richiedi la versione Dos ed esci se < 4.00
        reg.ax = &H3000
        InterruptX &H21, reg, reg
        IF (reg.ax AND &HFF) < 4 THEN EXIT SUB

        reg.bx = ASC(UCASE$(drive$)) - 64 ' codice del drive in BL
        reg.ds = VARSEG(buffer)            ' DS:DX deve puntare
        reg.dx = VARPTR(buffer)            ' al buffer di ricezione
        reg.ax = &H6900                    ' (serivzio non documentato)
        InterruptX &H21, reg, reg          ' chiamma il Dos
        ' il numero seriale è una doppia word all'offset 2 nel buffer
        ' e deve essere convertito nel formato mostrato dal comando DIR
        id& = CVL(MID$(buffer, 3, 4))
        serial$ = RIGHT$("0000000" + HEX$(id&), 8)
        serial$ = LEFT$(serial$, 4) + "-" + RIGHT$(serial$, 4)
        ' l'etichetta è una stringa di 11 caratteri all'offset 6
        label$ = MID$(buffer, 7, 11)
END SUB
```

9

IL COMPILATORE E IL LINKER

Programmare in BASIC non significa soltanto combinare più righe di codice per risolvere un problema; per ottenere il massimo dal linguaggio è anche necessario conoscere bene gli strumenti che permettono di trasformare i programmi sorgenti in file eseguibili: sto parlando, ovviamente, del compilatore e del linker e degli altri programmi e file di contorno a questi.

IL COMPILATORE BC.EXE

Il compito del compilatore BC.EXE è di leggere un file sorgente, solitamente con estensione BAS, e di generare il file oggetto OBJ corrispondente; questo file OBJ conterrà la traduzione delle istruzioni BASIC originarie in linguaggio macchina, l'unico linguaggio realmente eseguibile dal microprocessore Intel 80x86 in dotazione ai computer Ms-Dos. Oltre alle istruzioni in linguaggio macchina, il file oggetto contiene anche altre informazioni, tra cui i nomi delle procedure esterne a cui si fa riferimento nel programma e le cui chiamate non è stato possibile "risolvere" durante la compilazione: si ricordi che il compilatore esamina un modulo BASIC per volta, per cui non può essere in grado di calcolare la destinazione delle istruzioni CALL che fanno riferimento a routine presenti in altri moduli oppure a routine della libreria runtime del BASIC, come ad esempio PRINT o OPEN.

Normalmente il compilatore è richiamato dall'interno dell'ambiente di sviluppo QB/QBX/VBDOS usando il comando *Make Exe File*, ma in alcuni casi può essere conveniente sapere come eseguirlo direttamente dal prompt del Dos. La sintassi del compilatore è la seguente:

```
BC [options] sourcefile [,objectfile] [,listfile] [;]
```

dove **options** sono i vari switch che permettono di modificare il funzionamento di default del compilatore e **sourcefile** è ovviamente il file sorgente da compilare (l'estensione BAS può essere omessa). Gli altri argomenti sono opzionali: **objectfile** è il nome del file oggetto da creare (che normalmente ha estensione OBJ) e **listfile** è il file di listing prodotto dalla compilazione (con estensione LST, anche se per default non viene creato alcun file di listing); se la linea di comando termina con un punto e virgola il compilatore non richiede all'operatore gli argomenti eventualmente mancanti; in caso contrario il programma sospenderà l'esecuzione per richiedere interattivamente tutti gli argomenti omessi sulla linea di comando. La forma più semplice di questp comando è la seguente:

```
BC sourcefile ;
```

che crea un file oggetto con lo stesso nome del file **sourcefile** e con estebsione OBJ; se non viene specificata l'opzione **/O** il compilatore creerà un programma che dovrà funzionare con la libreria runtime BRUN.EXE (in QuickBASIC) o una delle tante varianti fornite con il BASIC PDS o il Visual BASIC. Per creare un programma eseguibile *stand-alone* occorre dare il comando

```
BC /o sourcefile ;
```

L'opzione **/t** indica al compilatore di omettere alcuni messaggi di copyright, ed è infatti usata dal BASIC per le compilazioni dall'interno dell'ambiente di sviluppo. L'uso di un file di listing è molto interessante per studiare come le varie istruzioni BASIC sono state tradotte in Assembly dal compilatore, grazie all'uso congiunto con l'opzione **/a**:

```
BC /o/a sourcefile, , listfile ;
```

Al termine della compilazione il compilatore mostra alcuni messaggi, simili ai seguenti:

```
42750 bytes avalaible
13940 bytes free
    0 Warning error(s)
    0 Severe error(s)
```

Il messaggio "bytes avalaible" indica la quantità di memoria disponibile al compilatore per il suo compito, mentre "bytes free" indica la quantità di memoria ancora libera al termine della compilazione: si ricordi che il compilatore deve conservare da qualche parte tutte le informazioni sul programma sorgente (la posizione delle etichette, i nomi delle variabili, ecc.), ed è possibile che un programma troppo grande diventi troppo complesso per poter essere processato da BC.EXE. Queste quantità non hanno quindi alcuna

relazione con l'uso della memoria da parte del programma BASIC trasformato in file oggetto.

Quando ci si accorge che la quantità di memoria libera si avvicina troppo allo zero si rende necessario suddividere il file sorgente in due o più moduli separati. Uno degli errori in cui si incorre abbastanza frequentemente, abituati e viziati come siamo dall'interprete del BASIC, è di considerare "completato" un programma che funziona all'interno dell'ambiente di sviluppo. Accade invece che un programma di grosse dimensioni funzioni perfettamente in forma interpretata ma non possa essere compilato senza provocare un errore di "Out of memory". Il motivo è di ordine tecnico: all'interno dell'ambiente ogni procedura separata ha a disposizione un intero segmento (cioè 64K) per il codice; nei programmi compilati, invece, un singolo modulo deve essere contenuto in un unico segmento di codice, e non può quindi superare i fatidici 64K. Questo spiega come sia possibile che programmi di grosse dimensioni possano essere perfettamente funzionanti nell'ambiente di sviluppo, e debbano invece essere suddivisi in più moduli per essere compilati correttamente.

LE OPZIONI DI COMPILAZIONE

/A indica di includere nel file di listing le istruzioni Assembly generate da ciascuna istruzione BASIC; ovviamente perché questa opzione abbia effetto deve essere specificato anche il nome del file di listing nel campo appropriato

/Ah indica che il programma contiene degli array huge più grandi di 64K; gli array di questo tipo si estendono per più di un singolo segmento, per cui in generale è necessario calcolare l'indirizzo segmento+offset di ciascun elemento; per evitare di inserire questi calcoli direttamente nel codice prodotto dal compilatore (aumentando notevolmente la dimensione dei file eseguibili) il compilatore genera invece una chiamata ad una routine del BASIC che effettua tutti i calcoli necessari nonché i dovuti controlli. I programmi che usano questa opzione sono quindi più lenti, ma allo stesso tempo sono protetti da eventuali riferimenti ad elementi inesistenti (questo tipo di errore non è riconosciuto a runtime a meno di non specificare le opzioni /ah o /d). Per minimizzare l'impatto sulla velocità di esecuzione si può decidere di compilare con /ah soltanto i moduli del programma che richiedono realmente di accedere agli array huge, ed omettere questa opzione negli altri moduli (questa possibilità non è ottenibile con i programmi compilati dall'interno dell'ambiente di sviluppo)

/C:nnn questa opzione permette di specificare la dimensione del buffer di ricezione usato per le comunicazioni seriali; il valore di default è 512 byte.

Se il programma indirizza entrambe le porte seriali COM1 e COM2, il buffer viene suddiviso in due metà ed assegnato a ciascuna porta. Queste aree di memoria sono gestite come buffer circolari, in cui le routine di libreria inseriscono i caratteri ricevuti dalla porta seriale e il programma applicativo estrae i caratteri richiesti usando le istruzioni INPUT$ o GET. In realtà, poiché queste aree sono nella far memory, i dati in arrivo dalla porta seriale sono oggetto di una doppia bufferizzazione, in quanto il BASIC li copia in un buffer più piccolo in DGROUP, la cui dimensione è influenzata dal parametro LEN della istruzione OPEN che ha aperto la comunicazione seriale. Normalmente il buffer di default è dimensionato adeguatamente, e questa opzione dovrebbe essere necessaria soltanto se si prevedono alte velocità di trasmissione oppure se il programma applicativo non è in grado di processare i dati in arrivo ad una velocità sufficientemente alta.

/D questa opzione dovrebbe essere usata soltanto durante la fase di debug del programma, e rimossa quando si intende creare la versione finale, in quanto l'eseguibile risultante è sensibilmente più lento e più grande del necessario. L'opzione /d ha i seguenti effetti:

a) per ciascuna riga di codice è inserita una chiamata ad una routine che controlla se l'operatore ha premuto il tasto Ctrl-Break; in tal modo è possibile bloccare i programmi compilati che entrano in un loop infinito, ma al costo di un overhead di ben cinque byte per ciascuna riga di codice sorgente

b) tutti gli accessi agli elementi degli array avvengono tramite una routine, che controlla la validità dell'indice ed evita quindi di leggere (o peggio ancora, scrivere) un'area di memoria che non appartiene all'array; questa è la stessa routine invocata quando il programma è compilato con l'opzione /ah·

c) le istruzioni GOSUB e RETURN sono realizzate per mezzo di chiamate a routine anziché inserendo direttamente i codici Assembly corrispondenti; in tal modo il BASIC può controllare che una istruzione RETURN corrisponda effettivamente ad una GOSUB in sospeso, evitando salti incontrollati agli indirizzi che casualmente si trovano sullo stack in quel momento

d) dopo tutte le operazioni aritmentiche che coinvolgono i valori INTEGER e LONG il compilatore inserisce una istruzione Assembly INTO (Interrupt on Overflow), che testa la condizione di overflow e se necessario salta ad una routine che mostra l'errore e interrompe l'esecuzione del programma

e) (solo in BASIC PDS e VBDOS Professional Edition) dopo le istruzioni DELETE, INSERT, UPDATE e CLOSE del gestore ISAM viene aggiunto un checkpoint.

/E questa opzione deve essere specificata se il programma usa il comando ON ERROR e RESUME seguita dal nome di una etichetta o da un numero di riga; per tenere traccia della posizione a cui saltare in caso di RESUME il compilatore genera quattro byte addizionali per ciascuna etichetta o numero di linea, ed inoltre i comandi GOSUB e RETURN sono realizzati per mezzo di chiamate a routine (come per /d), in modo da poter gestire correttamente gli errori anche nel mezzo di una sobroutine richiamata con GOSUB. Si veda anche la descrizione di /x.

/Es (solo Visual BASIC per Dos) indica di condividere la memoria espansa con routine scritte in altri linguaggi; questa opzione è necessaria nei programmi che sfruttano i form, che per default usano la memoria espansa per migliorare le prestazioni; questa opzione indica di salvare lo stato del driver EMS prima della chiamata ad una routine non-BASIC, e di ripristinarlo al rientro nel programma, e di fatto rallenta l'esecuzione di queste routine

/FPa (disponibile solo in BASIC PDS e in VBDOS Professional Edition) indica al compilatore di includere la libreria alternativa (*Floating Point Alternate*) per i calcoli in virgola mobile; questa libreria accelera sensibilmente i calcoli sui numeri SINGLE e DOUBLE quando il programma è eseguito su un sistema senza coprocessore matematico, al costo di una precisione leggermente ridotta. La maggiore velocità è possibile sia perché le routine di questa libreria sono più veloci di quelle della libreria standard, sia perché il programma eseguibile non deve perdere tempo a controllare l'esistenza del coprocessore. In generale, data la sempre maggiore diffusione del coprocessore matematico e dei computer basati sul microprocessore 80486 o Pentium (che incorporano un coprocessore), questa opzione dovrebbe essere evitata, a meno di non sapere per certo che il programma sarà eseguito su una macchina senza coprocessore e che siamo in grado di accettare la minore precisione dei calcoli.

/FPi (disponibile solo in BASIC PDS e in VBDOS Professional Edition) indica di includere le libreria matematiche che emulano il coprocessore; questa è la condizione di default, per cui questa opzione è in realtà ridondante.

/Fs (disponibile solo in BASIC PDS) specifica che il programma deve essere compilato per utilizzare le *far string*; in Basic PDS i programmi sono per default compilati per le *near string*. Si noti che tutti i moduli di un programma devono essere compilati con la medesima opzione per le stringhe e non è possibile compilare solo alcuni moduli con /Fs; questo tipo di errore può avvenire soltanto compilando manualmente dal prompt del Dos, e provoca solitamente il crash del sistema; quando si compila dall'interno dell'ambiente di sviluppo questa opzione è obbligatoria se è stata caricata una Quick Library. Le far string rallentano leggermente la velocità di esecuzione, e dovrebbero essere evitate a meno che il programma compilato senza /Fs generi un errore

"Out of string space"; in QuickBASIC sono permesse solo le near string, mentre in Visual BASIC per Dos tutte le stringhe sono far, per cui questa opzione è inutile.

/G2 (disponibile solo in BASIC PDS e VBDOS Professional Edition) indica al compilatore di creare codice binario che può essere eseguito solo su sistemi con processore 80286 o superiore. I programmi compilati con questa opzione sono leggermente più veloci e compatti, soprattutto nelle chiamate a subroutine con argomenti; infatti uno dei vantaggi del 80286 sui processori della generazione precedente è di poter eseguire il PUSH non solo di un registro ma anche del contenuto di una locazione di memoria, e quindi del valore di una variabile INTEGER oppure del descrittore di una stringa

/G3 (disponibile solo in VBDOS Professional Edition) è simile al precedente ed indica di generare codice eseguibile con microprocessore 80386 o più recente; in questo caso l'effetto è ancora più evidente, in quanto oltre alle ottimizzazioni comuni all'opzione /G2 sono anche ottimizzate le operazioni con gli interi LONG, che possono essere risolte con un'unica istruzione Assembly invece di una sequenza di istruzioni o una chiamata ad una subroutine della libreria runtime.

/Ib:nnn (disponibile solo in BASIC PDS e VBDOS Professional Edition) modifica il numero di pagebuffer usati dal gestore ISAM, dove ciascun pagebuffer occupa 2K; i valori di default sono 6 pagebuffer (12K) per PROISAM e 9 pagebuffer (18K) per PROISAMD, a cui occorre aggiungere circa 5K per i dati usati da PROISAM e circa 15K per i dati usati da PROISAMD. I valori di default rappresentano la minima quantità di memoria necessaria per poter utilizzare il gestore ISAM, e se possibile dovrebbero essere incrementati per ottenere le migliori prestazioni. La memoria per i pagebuffer è prelevata prima dalla memoria espansa EMS e, se questa non è sufficiente, la parte rimanente è allocata in memoria convenzionale. Il valore massimo per l'argomento è 512, che però è valido soltanto se il sistema dispone di memoria espansa.

/Ie:nnn (disponibile solo in BASIC PDS e VBDOS Professional Edition) indica di riservare il valore indicato di Kbyte in memoria EMS e di non renderlo disponibile al gestore ISAM; per default ISAM alloca per se fino a 1.2M di memoria EMS; questa opzione è in pratica necessaria solo se il programma fa anch'esso uso della memoria espansa.

/Ii:nnn (disponibile solo in BASIC PDS e VBDOS Professional Edition) specifica il numero di indici ISAM usati dal programma; il valore indicato deve essere minore di 500, il default è 30; questa opzione deve essere usata solo se il programma usa più di 30 indici non nulli.

/Lp (solo in BASIC PDS) genera un file eseguibile in modo protetto in OS/2; se il compilatore è eseguito sotto OS/2 il BASIC per default compila per il

modo protetto, quindi questa opzione è necessaria solo se stiamo compilando in Dos per un sistema funzionante con OS/2

/Lr (solo in BASIC PDS) genera un file eseguibile in modo reale; se il compilatore è eseguito in Dos oppure in una sessione in modo reale sotto OS/2 per default BC.EXE compila per il modo reale, quindi questa opzione è necessaria soltato se stiamo compilando in modo protetto in OS/2 per un sistema Dos

/MBF indica che il programma deve operare su file random mediante le funzioni CVS, CVD, MKS$ e MKD$ usando il formato proprietario Microsoft, che è differente dal formato IEEE usato a partire dal QuickBASIC 4.0; questa opzione permette di ricompilare senza modifiche un programma scritto per una precedente versione del BASIC e che deve operare su file di dati esistenti; a parte questo caso, l'opzione è superflua, in quanto il BASIC mette anche a disposizione le funzioni CVSMBF e CVDMBF, che permettono di leggere e convertire file scritti nel vecchio formato.

/O indica che il compilatore deve generare un file eseguibile stand-alone; senza questa opzione BC.EXE usa l'impostazione di default, e crea un eseguibile che richiede la presenza del runtime

/Ot (solo BASIC PDS) ottimizza l'esecuzione delle SUB, FUNCTION e DEF FN, evitando che all'inizio dell'esecuzione di una procedura il BASIC richiami una routine che memorizza lo stato attuale dello stack con lo scopo di poterlo ripristinare correttamente anche se l'uscita dalla procedura avviene all'interno di una routine richiamata con GOSUB; questa opzione velocizza l'esecuzione ma non riduce la dimensione del codice, sostituendo byte usati per la chiamata con altrettanti NOP; inoltre il BASIC disattiva questa opzione se nella procedura vi sono delle istruzioni GOSUB. Questa opzione non è stata inclusa in VBDOS in quanto il compilatore accluso ottimizza automaticamente il codice prodotto quando è possibile

/R memorizza le matrici per righe; per default il BASIC memorizza le matrici multi-dimensionali per colonne, ossia assegnando a locazioni di memoria consecutive prima tutti gli elementi del tipo (1,x), poi (2,x), e così via; al contrario, il linguaggio C memorizza gli array riga per riga, e questa opzione è utile se il programma BASIC deve interfacciarsi con una routine scritta in questo linguaggio.

/S scrive le costanti stringa nel codice oggetto anziché nella tavola dei simboli del compilatore, ed in tal modo evita un errore per memoria insufficiente durante l'esecuzione del compilatore; per comprendere questa opzione occorre sapere che il compilatore tiene conto di tutte le costanti stringa incontrate, individuando eventuali stringhe duplicate in modo da memorizzarle una volta soltanto nel codice; questa opzione indica invece di non tenere traccia delle stringhe incontrate.

Un effetto non documentato dell'opzione /S è di aggiungere due brevi routine (sedici byte in tutto) nel modulo OBJ creato dal compilatore; queste routine agiscono da dispatcher per le routine di assegnamento e concatenazione di stringhe nella libreria runtime, che quindi possono essere richiamate con una CALL a 16 bit anziché 32 e accettano gli argomenti nei registri anziché sullo stack; per questo motivo una operazione di questo tipo richiede meno byte del solito (anche se è leggermente meno veloce, a causa del salto addizionale necessario) e se il programma non include stringhe duplicate è possibile che in effetti l'opzione /S generi eseguibili più piccoli del normale, in contrasto con quanto dichiarato nella documentazione Microsoft ufficiale.

/T evita che il compilatore mostri gli avvertimenti (warning); questa opzione è usata quando si compila un programma con il comando Make Exe File dall'interno dell'ambiente di sviluppo; l'opzione è disponibile a partire da QuickBASIC 4.0 anche se è stata documentata soltanto a partire dal BASIC PDS.

/V inserisce dopo ogni istruzione del programma compilato una chiamata ad una routine centrale che controlla gli eventi COM, PEN, STRIG, TIMER, PLAY, KEY e UEVENT; questa opzione crea programmi di maggiori dimensioni (5 byte in più per ogni istruzione) e sensibilmente più lenti.

/W è simile alla opzione /V, con la differenza che il controllo avviene soltanto dopo ogni etichetta o numero di linea, per cui il rallentamento dei programmi è meno marcato.

/X indica la presenza di una istruzione ON ERROR con RESUME, RESUME NEXT o RESUME 0; la presenza di questa opzione fa crescere le dimensioni del programma compilato in ragione di quattro byte per ogni riga di BASIC, e inoltre rallenta il programma, per cui in generale dovrebbe essere evitata, usando se possibile l'istruzione RESUME seguita da un numero di linea (che richiede /E e aggiunge meno overhead) oppure facendo completamente a meno della istruzione ON ERROR

/Z (solo BASIC PDS) questa opzione è usata molto raramente e serve in pratica soltanto quando si compila il programma dall'interno del Microsoft Editor; il suo effetto è di includere nel file oggetto le informazioni sui numeri di riga, in modo da fornire all'editor la possibilità di posizionarsi sulle righe che hanno provocato errori di compilazione.

/Zd questa opzione è usata molto raramente e serve solo quando si intende eseguire il debug del programma usando il SYMDEB (Symbolic Debugger) della Microsoft, che era incluso nella versione 4.0 del MASM ma che attualmente non è più diffuso ed è stato di fatto sostituito col CodeView.

/Zi include nel file oggetto le informazioni necessarie ad utilizzare il CodeView; ovviamente questa opzione dovrebbe essere usata soltanto nelle fasi di test del programma, in quanto rallenta la compilazione e crea eseguibili di grandi dimensioni

Si noti che compilando dall'interno dell'ambiente di sviluppo le opzioni /E, /T, /V e /X sono incluse automaticamente se necessario; le opzioni /AH, /C e /MBF sono aggiunte solo se erano state specificate al momento di lanciare il programma QB/QBX/VBDOS; le rimanenti opzioni devono essere specificate agendo sulla dialog box oppure compilando il programma dal prompt del Dos.

Nel compilare un programma manualmente può essere conveniente sapere che il compilatore cerca i file di include (direttiva $INCLUDE) nella directory corrente ed in tutte le directory listate nella variabile d'ambiente INCLUDE, eventualmente separate dal punto e virgola; quindi, se conserviamo i file *.BI in una directory differente può essere necessario impostare questa variabile, ad esempio dando il seguente comando dal prompt del Dos

```
SET include=c:\bc7\include;c:\bc7\lib
```

(si noti il punto e virgola per separare le directory, esattamente come avviene per la variabile d'ambiente PATH); questa variabile viene impostata temporaneamente dall'ambiente QB/QBX/VBDOS prima di lanciare una compilazione con Make Exe File.

IL LINKER LINK.EXE

Il linker è un programma che processa il file oggetto creato dal compilatore (o i file oggetto, nel caso di un programma multi-modulo) e risolve tutti i riferimenti a routine non presenti nel file stesso, che potrebbero essere procedure scritte in BASIC ma presenti in altri moduli, procedure scritte in altri linguaggi e presenti sotto forma di librerie aggiuntive, oppure le procedure che eseguono i comandi del BASIC come PRINT e OPEN e che sono contenute nelle librerie runtime fornite con il linguaggio. Il linker fornito con il BASIC permette inoltre di creare le Quick Library, nella versione per BASIC PDS e VBDOS Professional Edition è anche in grado di creare moduli di overlay per programmi troppo grandi per poter essere caricati nella memoria convenzionale.

Come accade per il compilatore, la maggior parte dei programmatori BASIC vede il linking come un processo automatico che segue la compilazione richiamata per mezzo del comando Make Exe File impartito all'interno dell'ambiente di sviluppo; in realtà in alcuni casi è necessario sapere eseguire il linking "manualmente" per approfittare di alcune sue caratteristiche avanzate (ad es. gli overlay). La sintassi del comando LINK è la seguente

```
LINK [options] objfile [...],[exefile],[mapfile],[libfile...][;]
```

dove options sono le opzioni di linking, objfile è il file oggetto da linkare (i puntini stanno ad indicare che vi possono essere un numero qualsiasi di file di questo tipo sulla riga di comando, spazio permettendo), exefile è il nome del file eseguibile da creare (l'estensione EXE può essere omessa, e se l'intero argomento è omesso il default è il nome del primo file OBJ della lista), mapfile è un file di map che serve se intendiamo eseguire il programma con un debug, libfile sono una o più librerie (con estensione di default LIB); il punto e virgola finale, se presente, indica di non fermare l'elaborazione per richiedere le informazioni eventualmente mancanti.

Si noti che il primo file del campo objfile è quello che sarà eseguito appena caricato il programma; in tutti i file ad eccezione del primo il codice contenuto a livello di modulo (non appartenente cioè ad una SUB o FUNCTION) sarà incluso nel programma finale ma non potrà mai essere eseguito, a meno che non si tratti di una routine invocata in caso di errore e a cui punta una istruzione ON ERROR; per questo motivo, prima di linkare un programma multi-modulo, conviene controllare che tutti i moduli secondari non contangano codice a livello di modulo, a parte un eventuale gestore di errori. Nel campo objfile è anche possibile listare dei file di libreria (ma non come primo file), a condizione di specificare l'estensione LIB: attenzione però, perché listando una libreria in questo campo anziché alla fine della riga di comando si ottiene di includere l'intera libreria nel file eseguibile, e non le sole routine necessarie al programma. La versione 4.0 del QuickBASIC contiene un incredibile bug, che appunto lista il nome di una eventuale libreria aggiuntiva in questo campo anziché alla fine della riga di comando, con lo spiacevole risultato di includere l'intera libreria nel file eseguibile e aumentandone le dimensioni di conseguenza.

Il file di mapping include i nomi dei segmenti, la dimensione dello stack e altre informazioni che in generale non sono di grande utilità al programmatore BASIC, a parte forse quelle che riguardano l'utilizzo del segmento DGROUP da parte delle variabili semplici e degli array statici. Nonostante questa sua scarsa utilità, il linker assume per default che intendiamo creare un file di map (con stesso nome del file eseguibile ed estensione MAP), per cui per evitare di sprecare spazio su disco e rallentare inutilmente il processo di linking non è sufficiente omettere il nome del file di map ma occorre specificare il device NUL:

```
LINK myprog, , nul, mylib ;
```

È importante usare la versione del linker fornita a corredo del compilatore BASIC, in quanto è l'unica che sicuramente supporta tutte le opzioni necessarie. Questa considerazione è particolarmente importante per chi programma in più linguaggi e dispone di differenti versioni del linker sul proprio hard disk: se il linker non si trova nella directory corrente e il PATH di sistema è tale da

dare la precedenza ad una differente versione del linker, è possibile che il processo di linking non vada a buon fine e non si riesca a capirne il motivo.

Il linker può anche essere eseguito in modo interattivo, facendo in modo che sia il programma stesso a richiedere gli argomenti necessari e mancanti: in tal caso, però, è indispensabile omettere il punto e virgola finale:

```
LINK myprog
```

Quando LINK.EXE non riesce a trovare un file o una libreria l'elaborazione si arresta con un avvertimento (a meno di non usare l'opzione /B, vedi più avanti) ed il prompt "Enter new file spec:"; in tal caso occorre specificare il percorso completo del file non trovato e non soltanto la directory in cui si trova

```
Enter new file spec: c:\libs\mylib
```

Un comportamento non documentato permette di risparmiare qualche battuta di tasto, e scrivere solo la directory in cui si trova il file, a condizione di aggiungere un backslash in coda

```
Enter new file spec: c:\libs\
```

Omettendo il backslash finale il linker cercherebbe erroneamente un file LIBS.LIB nella directory radice di C:. E' possibile memorizzare una volta per tutte alcune opzioni usate comunemente nella variabile d'ambiente LINK; inoltre il linker ricerca tutte le librerie nella directory corrente e in tutte le sottodirectory contenute nella variabile d'ambiente LIB e separate da un carattere ";".

Se occorre linkare numerosi file oggetto oppure includere più librerie, è possibile che la riga di comando ecceda i 127 caratteri, la lunghezza massima dei comandi impartibili dal prompt del Dos. Per ovviare a questo limite il linker permette di raccogliere i comandi necessari (o parte di essi) in un *response file*, e indicare il nome di tale file usando il carattere "@". Nel response file i nomi dei file OBJ e LIB devono apparire nel medesimo ordine con cui li richiede il linker in modo interattivo, con l'unica differenza che è necessario usare il carattere "+" per indicare che la lista continua al rigo seguente. Ecco un esempio di response file, con accanto alcuni commenti:

```
/SE:512 myprog +        (le opzioni vanno per prime)
object2 +               (il simbolo + permette di continuare
object3 +               il campo dei file oggetto)
nocom                   (l'ultimo file non deve essere seguito da +)
                        (INVIO per accettare il default MYPROG.EXE)
nul                     (il file di map)
87 bcl7lenr ;           (la lista delle lib, evita ulteriori prompt)
```

Il response file precedente si usa in questo modo

```
LINK /NOE @respose.ext
```

da cui risulta evidente che è anche possibile aggiungere delle opzioni sulla riga di comando al di fuori del response file.

OVERLAY

Il linker fornito con BASIC PDS e Visual BASIC Professional Editioin è anche in grado di creare eseguibili con overlay. Il concetto di overlay è abbastanza semplice: alcuni dei moduli che compongono il programma non sono caricati in memoria alla partenza dell'applicativo, ma soltanto quando il programma principale richiama una delle routine contenute nel modulo stesso. Poiché gli overlay contengono codice e non dati - e quindi il loro contenuto non si modifica nel corso del programma - essi possono essere "scartati" quando termina l'esecuzione della routine, e ricaricati da disco quando si rende nuovamente necessario. In realtà è anche possibile usare la memoria espansa come "buffer" per non rileggere ogni volta il disco, e in tal caso il processo è praticamente istantaneo.

Per indicare al linker che uno o più moduli devono essere gestiti come overlay è sufficiente racchiudere il loro nome tra parentesi, come in

```
LINK myprog modulo1 (modulo2)  (modulo3 modulo4) ;
```

come si vede, non è detto che tutti i moduli secondari debbano essere sistemai negli overlay (vedi **modulo1** nell'esempio), ed inoltre è possibile raggruppare due o più moduli in un unico overlay. Tutti gli overlay sono caricati nella medesima regione di memoria, quindi un solo modulo di questo tipo alla volta può essere attivo; non è possibile specificare lo stesso modulo in differenti overlay e ovviamente non è possibile che il primo file specificato sulla riga di comando sia contenuto in un overlay. Gli overlay non possono superare la dimensione di 256K o 64 moduli distinti; i programmi con overlay non devono essere linkati con l'opzione /PAC.

Nel preparare un programma con overlay si devono seguire alcune strategie elementari. Ad esempio, le routine di carattere generale, ad esempio le routine di menù o di windowing, non dovrebbero essere inserite in un modulo overlay, poiché esse sono richiamate da più punti del programma e il programma passerebbe più tempo a caricare overlay che a processare dati. L'ideale candidato per gli overlay sono i moduli usati di rado, ad esempio quelli per le operazioni di servizio (ricostruzione indici, backup, ecc.) oppure per moduli di programma mutualmente esclusivi: si pensi ad esempio ad un programma di gestione con i moduli per la fatturazione, il magazzino, ecc.

LINK.EXE è in grado di produrre solo overlay interni, racchiusi nello stesso file EXE del programma principale; a differenza di quel che accade con un programma senza overlay, le chiamate CALL a 32 bit sono sostituite da una chiamata ad un particolare interrupt software (per default è l'interrupt 63, ma può essere modificato per evitare problemi di compatibilità), che attiva l'*overlay manager* (automaticamente incluso nel programma principale dal linbìker), il quale a sua volta provvede a caricare l'overlay appropriato.

L'overlay manager ricerca l'overlay dapprima nella directory corrente, poi in tutte le directory listate nel PATH di sistema; se anche questo tentativo fallisce viene richiesto all'operatore il percorso completo del file eseguibile.

LE OPZIONI DEL LINKER

Le opzioni del linker devono essere fornite sulla riga di comando prima del nome del primo file oggetto; come al solito, le opzioni sono precedute dalla barra "/", ma nel caso del linker possono essere abbreviate alle prime lettere che però non creino ambiguità; in altre parole, alcune lettere a destra di una opzione possono essere omesse purché l'abbreviazione ottenuta identifichi univocamente una ed una sola opzione. L'elenco che segue descrive tutte le opzioni del linker utili al programmatore BASIC e non include, o tratta molto superficialmente, le opzioni che non sono mai o quasi mai usate con questo linguaggio. La parte in minuscolo di ciascuna opzione indica i caratteri che possono essere omessi nell'abbreviazione dell'opzione stessa, ma si tenga presente che in generale le opzioni possono essere scritte indifferentemente in minuscolo o in maiuscolo, con la sola eccezione della opzione /r.

/Batch

Indica al linker che esso viene eseguito da un file batch, e che non deve quindi interrompere l'elaborazione se non riesce a trovare un file o una libreria specificata sulla riga di comando; in tal caso il linker emetterà dei messaggi di errore. In generale questa opzione dovrebbe essere evitata, anche quando si compila con un file batch, perché non fornisce la possibilità di rimediare all'errore. L'opzione /B non evita però che il linker richieda gli argomenti mancanti (usare il punto e virgola per questo scopo); un effetto secondario di questa opzione è di evitare la visualizzazione del messaggio di copyright e dell'eco del contenuto dei response file.

/COdeview

Prepara un file eseguibile per essere eseguito sotto il debugger Microsoft Codeview; in questo caso i file oggetto devono essere stati compilati con l'opzione /Zi; i file generati in questo modo sono molto più grandi del necessario, per cui questa opzione dovrebbe essere omessa nella creazione del programma definitivo; si noti, comunque, che i dati addizionali sono aggiunti in coda al file eseguibile vero e proprio, e non vengono quindi caricati

in memoria durante l'esecuzione, per cui il programma compilato con /CO dispone della stessa memoria libera del programma normale

/DOsseg

Forza un particolare ordine dei segmenti; questa opzione può essere utile soltanto se si linkano moduli scritti in altri linguaggi.

/Exepack

Indica al linker di rimuovere le sequenze di caratteri ripetuti e di ottimizzare lo spazio usato dalla tavola di rilocazione, ossia la tavola di informazioni usate dal Dos per caricare il programma a qualunque indirizzo di memoria e risolvere a run-time tutti i riferimenti di segmento. I file eseguibili prodotti da questa opzione sono in genere più corti del solito e sono quindi caricati più velocemente, anche se in alcuni casi il file "compresso" potrebbe risultare più lungo dell'originale (LINK emette un avvertimento in questo caso). I file più idonei ad essere linkati con questa opzione sono quelli che contengono array statici (che sono inizializzati a zero dal compilatore) oppure stringhe contenenti caratteri uguali, anche se in questo caso è preferibile usare direttamente una funzione STRING$ nel programma sorgente, che sfrutta meglio la memoria in DGROUP. Il livello di compressione ottenibile con questa opzione è comunque abbastanza basso, e se è davvero importante ridurre l'occupazione su disco dei file eseguibili è preferibile usare un compressore dinamico come PKLite della PKWare oppure LZEXE.

/Farcalltranslation

Questa opzione è in grado di velocizzare il programma eseguibile trasformando le CALL a 32 bit che si riferiscono a locazioni che si trovano nello stesso segmento in CALL a 16 bit; se viene usata congiuntamente alla opzione /PAC è anche possibile ottenere un (minimo) risparmio sulla dimensione del file (un byte per ogni istruzione modificata).

Le CALL a 32 bit sono inserite dal compilatore nei programmi oggetto per risolvere le chiamate alle procedure presenti in altri moduli oppure alla libreria runtime del BASIC stesso; dopo aver linkato i vari moduli e le librerie che compongono il programma può accadere, e infatti accade spesso, che la destinazione delle CALL risulta in realtà distante meno di 32K in avanti o all'indietro, e potrebbe essere quindi raggiunta con una CALL inter-segmento a 16 bit. L'opzione /F indica appunto al linker di sostituire dove possibile le istruzioni

```
CALL FAR etichetta
```

con l'equivalente sequenza

```
NOP
PUSH CS
CALL NEAR etichetta
```

dove l'istruzione NOP (no-operation) serve solo per mantenere la lunghezza complessiva di cinque byte; nonostante si tratti di tre istruzioni anziché una soltanto, questa sequenza è più veloce della CALL originale, in quanto non modifica il contenuto del registro di segmento CS. L'istruzione NOP potrà essere poi scartata da una eventuale opzione /PAC.

Come avvertono i manuali del linguaggio, esiste un piccolo rischio associato a questa opzione; per ricercare le istruzioni CALL candidate a questa modifica, il linker ricerca nel programma i byte con valore pari a &H9A, che è appunto il codice delle CALL FAR, e poiché questa ricerca è limitata al segmento di codice non si corre il pericolo che il linker si confonda con i valori presenti nei segmenti di dati. L'unico potenziale problema si ha con le istruzioni ON GOTO/GOSUB, che memorizzano nel segmento di codice l'offset delle etichette di destinazione, per cui esiste una probabilità non nulla che uno di questi offset contenga casualmente il valore 9A esadecimale. Per questo motivo, se il programma contiene istruzioni GOTO/GOSUB calcolate è necessario testare l'eseguibile a fondo dopo averlo compilato con questa opzione.

/HElp

Mostra la lista di queste opzioni sullo schermo, ignorando ogni altra opzione eventualmente presente sulla riga di comando.

/INFo

Indica di visualizzare le informazioni sul processo di linking, mostrando quando comincia una nuova fase e i nomi dei file oggetto e delle libreria man mano processati. Queste informazioni vengono mostrate sullo schermo e non possono essere redirezionate usando il carattere ">".

/LInenumbers

Include i numeri di linea e la relativa posizione nel file di map; questa opzione era necessaria per eseguire il debug mediante l'obsoleto SYMDEB (ora sostituito da CodeView) e prevede che il file sorgente sia stato compilato con l'opzione /Zi, in caso contrario /LI non ha alcun effetto.

/Map

Include nel file di map la lista dei *simboli pubblici* definiti dal programma, ordinata per nome e per indirizzo; senza questa opzione il file di map contiene solo la lista dei segmenti usati dal programma. Questa opzione è utile solo in particolari occasioni programmando in Assembly o qualche altro linguaggio.

/NODefaultlib

/NODefaultlib:filename

Indica di non usare la libreria di default specificata nel file oggetto; questa opzione è necessaria in quanto per default il compilatore BC.EXE prevede di linkare il programma per l'uso di una libreria runtime come BRUN45.EXE (in QuickBASIC) o una delle librerie fornite a corredo del BASIC PDS e del Visual BASIC per Dos. E' possibile specificare dopo il carattere ":" il nome esatto della libreria da non linkare, nel caso in cui i file oggetto contegano più di un singolo nome di libreria; se si usa questa opzione è necessario specificare un nome di libreria nell'ultimo campo della riga di comandi fornita al linker.

/NOExtdictionary

Normalmente in un programma tutti i nomi dei simboli pubblici sono unici, definiti cioè solo una volta nell'intera applicazione; tali simboli sono i nomi delle variabili e delle procedure del programma BASIC, e i nomi delle routine contenute nella libreria runtime del linguaggio; man mano che procede la costruzione del programma eseguibile, il linker conserva in una tavola interna i nomi pubblici incontrati, e genera errore quando un simbolo è ridefinito. In alcuni casi, tuttavia, questa ridefinizione è perfettamente lecita (ad esempio quando usiamo uno stub file) per cui è necessario avvisare il linker di non emettere alcun errore. Questa opzione dovrebbe essere usata solo se realmente necessario, in quanto rallenta le operazioni del linker.

/NOFarcalltranslation

Disabilita la conversione delle CALL FAR; normalmente questa opzione è superflua, in quanto questa è la condizione di default, ma è presente per annullare una eventuale opzione /F nella variabile d'ambiente LINK.

/NOLogo

Indica a LINK.EXE di non mostrare il messaggio iniziale di copyright.

/NOPackcode

Annulla l'effetto di una opzione /PAC eventualmente presente nella variabile d'ambiente LINK.

/NOPACKFunctions

(solo Visual BASIC per Dos) annulla l'effetto di una opzione /PACKF eventualmente presente nella variabile d'ambiente LINK.

/Overlay:nnn

Per default il manager di overlay del BASIC usa l'interrupt software 63 (3F esadecimale), ma con questa opzione è possibile modificare questa impostazione nel caso sorgessere problemi di compatibilità con altro software o periferiche, oppure quando l'applicativo richiama con SHELL un altro programma BASIC anch'esso compilato con overlay; **nnn** può essere un numero decimale tra 0 e 255, un numero ottale tra 0 e 0377, o un numero esadecimale tra 0x00 e 0xFF.

/PACkcode:nnn

Questa opzione indica al linker di raggruppare se possibile moduli di codice differenti, riducendo il numero di segmenti distinti; questa opzione permette di creare programmi leggermente più corti e più veloci, a condizione che sia utilizzata anche l'opzione /F,. Il valore **nnn** è opzionale e indica la dimensione massima in byte dei segmenti (il valore di default è 65535, la dimensione massima di un segmento); questa opzione non deve essere usata se il programma include overlay, e può creare problemi di compatibilità con alcuni programmi Assembly.

/PACKData:nnn

Questa opzione è simile a /PAC ma funziona con i segmenti didati anziché i segmenti di codice; normalmente questa opzione può essere ignorata dai programmatori BASIC.

/PACKFunctions

(solo Visual BASIC per Dos) rimuove dal programma le funzioni non referenziate; questo è il comportamento assunto per default dal linker, per cui questa

opzione è in realtà inutile ed è stata prevista solo per disabilitare una eventuale opzione /NOPACKF presente nella variabile d'ambiente LINK.

/PAUse

Inserisce una pausa prima di scrivere il file eseguibile su disco; questa opzione è utile se si lavora senza hard disk e lo spazio disponibile su floppy è insufficiente.

/Quicklibrary

Genera una Quick Library anziché un file eseguibile; se questa opzione è presente il campo **exefile** indica il nome della Quick Library ed ha per default estensione QLB anziché EXE.

/r

(solo Visual BASIC per Dos) impedisce al linker di usare la memoria estesa; per essere riconosciuto deve essere il primo switch sulla riga di comando.

/SEgments:nnn

Imposta il numero massimo di segmenti; questa opzione si rende necessaria quando le librerie includono un gran numero di simboli (procedure e funzioni) e la memoria usata dal linker per le tavole interne risulta insufficiente; in pratica questa opzione risulta necessaria quando si utilizza una libreria prodotta da terze parti oppure quando si tenta di creare una Quick Library con molte procedure e funzioni. Il numero di default dei segmenti riservati dal linker è 128, il parametro **nnn** può variare da 1 a 3072.

/STtack:nnn

Imposta la dimensione dello stack in byte; **nnn** può essere un qualunque numero positivo minore di 65535 (espresso in decimale, ottale o esadecimale); la dimensione di default dello stack è 3K per i programmi Dos e 3.5K per i programmi OS/2. Si noti che la Microsoft scoraggia l'uso di questa opzione, in quanto il medesimo risultato può ottenersi con una istruzione CLEAR (in QuickBASIC 4.x) o meglio ancora STACK (in BASIC PDS e Visual BASIC per Dos).

GLI STUB FILE

Gli stub file offrono la possibilità di ridurre la dimensione del file eseguibile generato in fase di linking, sostituendo le routine di default della libreria del BASIC con altre routine con funzionalità minori e quindi più corte. In alcuni casi gli stub file non diminuiscono in alcun modo le caratteristiche del programma, e servono per evitare che il linker includa delle funzioni a cui non siamo interessati.

Per comprendere come funzionano gli stub file occorre pensare a come sono trattati i riferimenti alle funzioni di libreria del BASIC, e in generale tutti i riferimenti non risolti all'interno del medesimo file OBJ: per prima cosa il linker tenta di risolvere i riferimenti nei moduli OBJ listati nel primo campo della riga di comando, poi passa nell'ordine alle librerie dichiarate. A parte l'ordine con cui sono ricercati, l'unica differenza tra i riferimenti risolti in un file OBJ e quelli risolti in un file LIB è che nel primo caso tutto il file viene incluso nel programma finale, mentre nel secondo è inclusa solo la routine interessata e quelle ad essa correlata. Allora, facendo in modo che il riferimento ad un simbolo sia risolto in un OBJ si ottiene di fatto di evitare che sia linkata la routine del BASIC.

Gli stub file contengono alcuni dei simboli definiti nella libreria runtime del BASIC, a cui però corrisponde una procedura "nulla" oppure una routine più breve (e meno completa) di quella standard. Facendo precedere lo stub file alla libreria di default si ottiene di fatto di ridurre la dimensione del file eseguibile; nella maggior parte l'attivazione di una delle procedure nulle ha il solo effetto di provocare un messaggio di errore "Feature Stubbed Out". Gli stub file possono essere usati per modificare la libreria runtime standard oppure, più comunemente, nella compilazione dei singoli programmi che li richiedono.

Per poter usare uno o più stub file il programma deve essere linkato con l'opzione /NOI; in caso contrario, infatti, il processo termina con un errore indicando che uno o più simboli sono definiti più volte (una volta nello stub file e una volta nella libreria di default). Un esempio di uso di stub file

```
BC  /T/O myprog ;
LINK /NOE myprog nocom nofltin, ,nul, 87 bcl71efr ;
```

Ecco un elenco degli stub file forniti con le varie versioni del Microsoft BASIC:

NOFLTIN.OBJ (solo BASIC PDS e Visual BASIC) elimina il supporto per i numeri in virgola mobile dalle istruzioni INPUT, VAL e READ, che quindi accettano esclusivamente interi INTEGER e LONG.

NOEDIT.OBJ (solo BASIC PDS e Visual BASIC) riduce le capacità della istruzione INPUT; usando questo stub file sono operativi soltanto i tasti INVIO e BACKSPACE.

NOCOM.OBJ elimina il supporto delle porte seriali dalle istruzioni OPEN; se il programma comprende una istruzione OPEN il cui argomento è una variabile o una espressione, per default il BASIC include nel programma il supporto per le comunicazioni seriali, che accresce sensibilmente le dimensioni dell'eseguibile.

NOFORMS.OBJ (solo Visual BASIC) rimuove il supporto per i form e per le procedure ad eventi.

NOLPT.OBJ (solo BASIC PDS e Visual BASIC) elimina il supporto delle porte parallele dalle istruzioni OPEN e il supporto del tasto Ctrl-PrntScreen, che permette di attivare e disattivare a runtime l'eco sulla stampante di quanto appare sullo schermo.

NOEVENT.OBJ (solo BASIC PDS e Visual BASIC) rimuove il supporto per l'intrappolamento degli eventi; ha effetto solo se usato per ricostruire il modulo runtime.

NOEMS.OBJ (solo BASIC PDS) impedisce che un programma con overlay sfrutti la memoria espansa per conservare gli overlay, forzando invece lo swap su disco.

NOISAM.OBJ (solo BASIC PDS e Visual BASIC Professional Edition) rimuovo il supporto del gestore ISAM; ha effetto solo se usato per ricostruire il modulo runtime.

OVLDOS21.OBJ (solo BASIC PDS) include il supporto del Dos 2.1 nei programmi che usano gli overlay.

SMALLER.OBJ riduce la lunghezza dei messaggi di errore; usando questo stub file i programmi che terminano con un errore a runtime mostreranno solo il codice numerico dell'errore anziché la sua descrizione per esteso.

87.LIB rimuove le routine di libreria che forniscono l'emulazione software del coprocessore matematico; i programmi linkati con questo stub file richiederanno la presenza del coprocessore per funzionare correttamente.

NOTRNEMR.LIB (solo BASIC PDS) e **NOTRNEM.OBJ** (solo Visual BASIC per Dos) rimuovono il supporto per i comandi e le funzioni che usano le operazioni trascendentali: l'elevamento a potenza "^", ATN, CIRCLE usato per disegnare archi, i sottocomandi A e T della istruzione DRAW, le funzioni COS, EXP, LOG, SIN, SQR, TAN.

TSCNIONR.OBJ, TSCNIOFR.OBJ (solo BASIC PDS, rispettivamente da usarsi con le stringhe near e le stringhe far) e **TSCNIO.OBJ** (solo Visual BASIC) diminuiscono le dimensioni dei programmi che funzionano esclusivamente in modo testo escludendo il supporto per i caratteri di controllo.

NOGRAPH.OBJ (solo BASIC PDS e Visual BASIC) rimuove il supporto per tutti i comandi grafici e per le istruzioni SCREEN con argomento non nullo; questo file può essere usato per tutti i programmi che funzionano esclusivamente in modo testo.

Tutti gli stub file che seguono sono disponibili solo in BASIC PDS e Visual BASIC per Dos; essi sono superflui se è stato già incluso il file NOGRAPH.OBJ o uno dei file TSCNIO*.OBJ:

NOCGA.OBJ	rimuove il supporto per i modi grafici 1 e 2 (scheda CGA).
NOHERC.OBJ	rimuove il supporto per il modo grafico 3 (scheda Hercules).
NOOGA.OBJ	rimuove il supporto per il modo grafico 4 (scheda Olivetti)
NOEGA,OBJ	rimuove il supporto per i modi grafici 7-10 (scheda EGA)
NOVGA.OBJ	rimuove il supporto per i modi grafici 11-13 (scheda VGA).

Gli stub file che rimuovono il supporto per uno o più modi grafici sono necessari solo se il programma include una istruzione SCREEN *con una variabile o una espressione come argomento: non potendo sapere quale sarà il valore della variabile durante l'esecuzione, il compilatore* BC.EXE *include il supporto per* tutti *i modi grafici previsti dal* BASIC. *Per questo motivo le istruzioni* SCREEN *dovrebbero sempre specificate con un argomento costante; ad esempio, il seguente frammento di programma:*

```
PRINT "Indica il modo grafico desiderato (1) o (2) "
DO: i$ = INKEY$: LOOP UNTIL i$ = "1" OR i$ = "2"
SCREEN VAL(i$)
```

dovrebbe essere riscritto in questo modo:

```
PRINT "Indica il modo grafico desiderato (1) o (2) "
DO: i$ = INKEY$: LOOP UNTIL i$ = "1" OR i$ = "2"
IF i$ = "1" THEN SCREEN 1 ELSE SCREEN 2
```

QUICK LIBRARY

Le Quick Library sono una delle caratteristiche più utili del BASIC, permettendo di precompilare porzioni di codice oppure di rendere disponibili routine scritte in C o Assembly anche ai programmi interpretati all'interno dell'ambiente di sviluppo.

Prima di decidere se compilare o meno una serie di procedure in una Quick Library si tenga però presente quanto segue:

- il codice a livello di modulo contenuto in un file BASIC da trasformare in Quick Library non può essere eseguito (a meno che non si tratti di un gestore di errore o di una funzione DEF FN), ma occupa comunque spazio nella libreria; per questo motivo prima di trasformare il file in libreria e consigliabile cancellare queste istruzioni superflue
- le procedure di una Quick Library devono essere autonome, e non devono richiamare procedure e funzioni non comprese nella Quick Library stessa; in caso contrario il linker produrrà errore
- il debugger integrato del BASIC non può eseguire il trace durante la chiamata ad una procedura in una Quick Library; occorre quindi che le routine da convertire in libreria siano state testate a fondo
- sebbene in generale le Quick Library occupano meno memoria del programma sorgente corrispondente, è anche vero che - lavorando in BASIC PDS o Visual BASIC - i programmi BASIC sorgente possono essere memorizzati in memoria espansa; al contrario, una Quick Library è sempre caricata in memoria convenzionale

A proposito dell'ultimo punto, è alquanto strano che la Microsoft non abbia provveduto, in occasione del rilascio del Visual BASIC per Dos, a fornire la possibilità di caricare le Quick Library in alta memoria (la zona compresa tra il limite dei 640K e 1024)K. A dimostrazione del fatto che si tratta di una semplice dimenticanza, la SoftWhale fornisce una utility chiamata dSWAPPER che - tra le altre funzioni - permette di caricare una qualsiasi Quick Library in alta memoria, e libera una eguale numero di byte in memoria convenzionale; anche QBKIT è dotata di questa possibilità, ed è al momento l'*unica* libreria in commercio in grado di non sprecare neanche un byte in memoria convenzionale.

La generazione di una Quick Library composta di uno o più moduli BASIC è molto semplice, essendo previsto un apposito comando del menu **Run**. In molti casi, tuttavia, è necessario o preferibile eseguire questa operazione in modo "manuale", lanciando il linker dal prompt del Dos; questa operazione è in realtà molto semplice e risulta simile alla creazione di un qualsiasi eseguibile, con la sola differenza che è necessario specificare l'opzione /Q e fornire come libreria di supporto quella fornita a corredo della particolare versione del linguaggio (QBQLB in QuickBASIC, QBXQLB in BASIC PDS e VBDOSQLB in Visual BASIC per Dos)

```
LINK /q objfile [objfile2 ...], , nul, qbxqlb ;
```

In BASIC PDS è indispensabile che i vari file oggetto siano stati creati con l'opzione /Fs, in quanto le stringhe far sono le uniche disponibili ai programmi interpretati con cui devono colloquiare le Quick Library; se si dimentica questo particolare il linker non produrrà alcun messaggio di errore, ma al momento di caricare la Quick Library il sistema andrà immediatamente in crash.

Se già si dispone della libreria in formato LIB è molto semplice generare la Quick Library corrispondente; è sufficiente, infatti, listare il file LIB nel primo campo della riga di comando (senza omettere l'estensione)

```
LINK /q library.lib, , nul, qbxqlb ;
```

***Importante:** i file di supporto forniti con le versioni italiane del QuickBASIC e del Visual BASIC Standard Edition sono inspiegabilmente differenti da quelli forniti con le versioni americane, per cui accade che una Quick Library preparata con una versione non sia usabile con l'altra. Questo aspetto è particolarmente seccante, in quanto preclude a coloro che usano una versione italiana del linguaggio di usare alcune Quick Library prodotte da software d'oltreoceano o reperibili nel pubblico dominio. Il sistema appena illustrato permette di ovviare a questa limitazione.*

IL PROGRAMMA LIB.EXE

Oltre al compilatore e il linker, i programmatori BASIC farebbero bene ad impratichirsi con un altro programma di utilità fornito con il linguaggio: si tratta del gestore di librerie LIB.EXE, che permette di manipolare file in formato LIB estraendo, aggiungendo o sostituendo i singoli moduli che li compongono. La sintassi di LIB è la seguente:

```
LIB [options] libfile [commands...], [listfile], [outputlib] [;]
```

dove **libfile** è la libreria da processare (l'estensione LIB è assunta per default), **listfile** è un file di list opzionale, **outputlib** è il nome della nuova libreria da creare (se omessa, i comandi saranno applicati alla libreria originaria, e verrà creato un file con estensione BAK). I comandi accettati da LIB sono i seguenti

+modulo	aggiunge un modulo oggetto o una intera libreria
-modulo	rimuove un modulo oggetto
*modulo	estrae una copia di un modulo oggetto
-+modulo	sostituisce un modulo esistente con un nuovo modulo
-*modulo	estrae e rimuove un modulo

Quindi, per aggiungere un modulo ad una libreria esistente possiamo scrivere

```
LIB mylib +nuovomod ;
```

se il modulo non si trova nella directory corrente è anche possibile indicare un percorso:

```
LIB mylib +d:\objfiles\nuovomod ;
```

Il comando "*" permette di estrarre un modulo da una libreria (senza però

cancellarlo dalla libreria stessa), creando un file nella directory corrente o nella directory indicata dal comando

```
LIB mylib *d:\objfiles\module ;
```

LIB può anche essere molto utile senza eseguire alcun comando, solo per creare il listato di una libreria, in questo modo

```
LIB mylib,  library.lst ;
```

Il file ottenuto in questo modo contiene due liste correlate: i nomi delle procedure con accanto quello del modulo in cui si trovano, e i nomi dei moduli con le procedure che ciascuno contiene. Può essere molto istruttivo creare il list file della libreria runtime del BASIC e vedere come sono raggruppate le routine; in QuickBASIC il comando da impartire è il seguente:

```
LIB bcom45.lib, quick.lst ;
```

i simboli pubblici contenuti in questa libreria sono 1483, suddivisi in 187 moduli; il fatto che che vi siano evidentemente moduli che contengono numerose procedure (in alcuni casi più di trenta procedure per modulo) si esprime dicendo che la la libreria runtime del QuickBASIC ha una scarsa *granularità*; questo costituisce un problema, in quando è noto che richiamando una qualsiasi delle routine contenute in un modulo si ottiene di includere l'intero modulo nel programma eseguibile. Nel BASIC PDS e in Visual BASIC le cose vanno leggermente meglio, perché le librerie runtime sono più granulari, ma il problema rimane ed è alla base della convizione generale che il BASIC produca eseguibili di grosse dimensioni. Ovviamente l'ideale sarebbe di avere una corrispondenza uno ad uno (un procedura per ciascun modulo), ma per motivi pratici questo è impossibile, poiché molte procedure sono correlate e devono condividere subroutine e variabili di lavoro. Comunque, è abbastanza facile fare meglio del BASIC: ad esempio la libreria QBKIT che ho scritto per la SoftWhale conta oltre 550 routine Assembly suddivise in circa 490 moduli separati; di fatto, usando le istruzioni che QBKIT mette a disposizione in luogo delle istruzioni classiche del BASIC è possibile creare file eseguibili di dimensioni ridottissime.

Studiare con cura il file QUICK.LST non è affatto tempo sprecato, e si possono fare alcune scoperte molto interessanti, anche se i nomi delle routine della libreria sono in genere differenti dai nomi dei comandi BASIC corrispondenti e bisogna lasciarsi guidare dall'intuito. Ad esempio, il primo modulo della libreria ha nome **dkstmt** (presumibilmente sta per "Disk Statements") e contiene i seguenti simboli:

B$FILS   B$LOCK   B$KILL   B$NAME   B$REST

da cui si ricava l'informazione che usando in un programma uno dei seguenti comandi: FILES, LOCK, KILL e NAME, il linker include il supporto per tutti gli altri; allora, se il nostro programma usa uno soltato dei comandi precedenti, può essere più vantaggioso richiamare direttamente l'interrupt 21h del Dos

(come spiegato altrove in questo stesso testo) ed evitare l'inclusione dell'intero modulo, risparmiando così 552 byte. Più avanti nel listato i moduli **llegasup** (Ega Support) e **llvgasup** (Vga Support) ci forniscono l'informazione che il BASIC richiede ben 2128 byte per supportare la scheda grafica EGA e 1948 byte per la scheda VGA, e ci danno quindi una idea di quanti byte possiamo risparmiare linkando il programma con gli opportuni stub file.

Guardando il listato si nota subito che la maggioranza delle routine ha un nome che comincia per **B$**; questo serve ad impedire che il nome sia inavvertitamente usato per una procedura o funzione del programma sorgente (il carattere "$" non è ammesso nei nomi delle procedure BASIC); per lo stesso motivo alcuni nomi contengono uno o più caratteri di sottolineato. Una curiosità: nella libreria relativa al QuickBASIC 4.5 esiste una routine che non contiene alcun carattere proibito: si tratta della procedura MSRWAT, ed infatti è possibile compilare senza errori il seguente programma di un solo rigo

```
CALL msrwat
```

non chiedetemi cosa fa la routine, so soltanto che il programma provoca un blocco al sistema.

Anche il programma LIB supporta i *response file*, solo che in questo caso il carattere di continuazione della riga è "&".

COMPILARE E LINKARE USANDO I FILE BATCH

In questo capitolo abbiamo mostrato numerose possibilità del compilatore e del linker, molte delle quali sono accessibili soltanto compilando "manualmente" il file dal prompt del Dos e non dall'interno dell'ambiente di sviluppo. Vedremo ora come usare alcuni programmi batch per rendere più facile il processo. Anche se non mi pare sia documentato su qualche manuale, si può verificare facilmente che sia il compilatore BC.EXE che il linker LINK.EXE restituiscono un codice di errore al Dos, nullo se la compilazione o il linking sono andati a buon fine, e maggiore di zero in caso contrario.

E' possibile allora scrivere un breve file batch che compila un file sorgente e, se la compilazione ha avuto successo, procede a linkarlo con la libreria standard del linguaggio. Per semplicità, nell'esempio che segue faremo riferimento alla compilazione di file stand-alone (che non richiedono cioè il modulo runtime) in QuickBASIC 4.5, ma è ovvio che il discorso può essere immediatamente allargato alle altre versioni del linguaggio e alle altre opzioni di compilazione:

```
REM file COMPILE.BAT
BC /O/T %1 ;
IF NOT ERRORLEVEL 1 LINK %1, , nul, bcom45 ;
```

In questo modo le opzioni di compilazione e di linking sono fissate all'interno del file batch, ma è facile prevedere due variabili di ambiente BCCMD e LINKCMD in cui memorizzare le opzioni desiderate; il nostro programma batch diventa allora

```
REM file COMPILE2.BAT
BC /O/T %bccmd% %1 ;
IF NOT ERRORLEVEL 1 LINK %linkcmd% %1, , nul, bcom45 ;
```

Per i programmi multi-modulo è necessario un piccolo sforzo in più; ecco il programma batch che ho costruito per questo compito, più complicato del precedente ma anche molto più potente

```
REM file COMPILE3.BAT
REM
SET objlist=
:ripeti
ECHO Compilazione file %1...
BC /O/T %bccmd% %1 ; > error.lst
IF ERRORLEVEL 1 GOTO errore
SET objlist=%objlist% %1
SHIFT
IF NOT "%1"=="" GOTO ripeti
ECHO Linking...
LINK %linkcmd% %objlist%, , nul, bcom45 ; >error.lst
IF NOT ERRORLEVEL 1 GOTO fine
:errore
TYPE error.lst
:fine
```

E' allora possibile compilare un qualsiasi numero di moduli in questo modo:

```
COMPILE3 modulo1 modulo2 modulo3
```

Il programma precedente presenta alcune caratteristiche molto interessanti; prima di tutto l'output del compilatore e del linker è inviato al file ERROR.LOG anziché sullo schermo, e il contenuto di tale file è mostrato solo se necessario, in altre parole se è realmente avvenuto un errore: in caso di errore è preferibile avere una lista di errori su file per poterla analizzare con comodo, piuttosto che vederla scorrere via inesorabilmente sullo schermo. Man mano che il compilatore analizza i parametri sulla riga di comando, essi sono "accumulati" nella variabile d'ambiente OBJLIST in modo da potere usare poi il contenuto di quest'ultima nel richiamare il linker.

L'uso dei file batch per alleviare il lavoro del programmatore non si ferma qui; ad esempio, disponendo di un disco RAM è possibile scrivere un file batch che copia i programmi BC e LINK e le librerie runtime su tale disco (operazione che deve essere eseguita una sola volta all'accensione della macchina) e procede poi a compilazioni super-veloci. Ecco lo scheletro di un programma del genere, che assume che E:\ sia il disco virtuale:

```
REM programma FASTCOMP.BAT
REM
e:
IF EXIST bc.exe GOTO finecopia
ECHO Copia dei file di supporto su E: ...
```

```
COPY c:bc.exe >nul
COPY c:link.exe >nul
COPY c:bcom45.lib >nul
:finecopia
ECHO Compilazione dei file sorgenti...
REM
REM inserire qui le istruzioni che compilano i
REM file BAS che compongono il progetto, seguito
REM dalle istruzioni per il linking
REM
REM infine ricopia il file EXE ottenuto in C:, ad es.
COPY myprog.exe c: >nul
:fine
c:
```

Il vantaggio di questo approccio è di evitare di riempire il disco rigido C: di tutti i file OBJ creati dalle varie compilazioni.

COMPILAZIONI PIU' VELOCI

Lavorando con programmi multimodulo si scopre subito una delle limitazioni del comando **Make Exe File** dell'ambiente di sviluppo del BASIC, che ricompila sempre e comunque tutti i file di una applicazione anche se magari abbiamo apportato una modifica ad uno solo di essi; è evidente che in questo caso basterebbe ricompilare solo il file sorgente in questione, ottenendo un sostanziale risparmio di tempo. A dire il vero il BASIC PDS e il Visual BASIC Professional Edition offrono il programma NMAKE; si tratta di una una utility che permette di confrontare la data dei file sorgente ed oggetto e di compilare solo i file sorgenti la cui data risulta posteriore a quella del file oggetto con lo stesso nome (o per cui non esiste il file oggetto corrispondente); lo svantaggio di questo programma è di essere abbastanza macchinoso, in quanto occorre imparare un nuovo "linguaggio" per costruire i file di script, e che comunque costringe ad abbandonare l'ambiente QBX/VBDOS.

Che ne direste invece di un sistema per velocizzare il processo di compilazione & link dall'*interno* dall'ambiente di sviluppo ?

Prima di procedere, cerchiamo di capire cosa succede quando eseguiamo il comando **Make Exe File** del menù Run: per ciascun modulo di cui è composto il programma il compilatore BC.EXE è richiamato con gli appositi parametri, e solo se non vi sono errori viene richiamato il linker. Tutto semplice e lineare, quindi, ma molto poco efficiente, per due motivi:

1. come abbiamo detto, il BASIC ricompila *tutti* i moduli di cui il programma è composto, anche se il sorgente non è mai stato modificato dall'ultima compilazione

2. per quanto sia possibile influire su alcune opzioni di compilazione (tipo /AH oppure /Zi), la stessa possibilità non è offerta per il linker, per esempio non è possibile includere gli *stub file* come NOCOM o NOGRAPH

La soluzione del primo punto non sarebbe troppo complicata: basterebbe infatti che il BASIC controllasse la data dell'ultima modifica del sorgente e la confrontasse con quella del file OBJ corrispondente, evitando di ricompilare i sorgenti la cui data di modifica è precedente alla data di creazione del file oggetto corrispondente. Vediamo invece cosa è possibile fare con le nostre forze per migliorare la situazione.

Prima di tutto di occorre una funzione in grado di restituire la data e l'orario di un file; la routine **FileDateTime** chiama il Dos per ottenere queste informazioni, e le restituisce in un unico intero LONG, in modo da favorire i confronti tra le date di creazione di file differenti.

```
FUNCTION FileDateTime& (filename$)
     ' Restituisce la data e l'orario di un file come intero LONG
     ' oppure -1 se il file non esiste o vi è qualche tipo di errore
     '    (richiede Basic PDS o VBDOS per la presenza di ON LOCAL ERROR)
     DIM reg AS RegTypeX, filenum%, fdate&, ftime&
     ON LOCAL ERROR RESUME NEXT

     filenum% = FREEFILE
     OPEN filename$ FOR INPUT AS filenum%
     IF ERR THEN FileDateTime& = -1: EXIT SUB

     reg.ax = &H5700                ' servizio Dos che legge la data e l'ora
     reg.bx = FILEATTR(filenum%, 2)' BX = handle del file
     InterruptX &H21, reg, reg
     CLOSE #filenum                 ' chiudi il file

     ' il Dos restituisce la data in DX e l'orario in CX
     ' nelle righe seguenti questi dati sono combinati per formare un
     ' unico intero a 32 bit - il bit meno significativo dell'orario
     ' è troncato, e la data è shiftata verso sinistra di 15 bit
     ' il bit più significativo è sempre zero, per cui il risultato è sempre >0

     fdate& = (reg.dx AND &HFFFF&)     * &H8000&
     ftime& = (reg.cx AND &HFFFE&)     \ 2
     FileDateTime& = fdate& OR ftime&
END FUNCTION
```

Dato che il BASIC tratta solo con gli interi con segno, l'unico modo per non complicare troppo il programma è di trattare solo con numeri positivi, imponendo cioè che il bit più a sinistra (il bit del segno) sia zero; d'altra parte noi intendiamo "impacchettare" due interi da 16 bit e quindi l'unico modo di lavorare con soli numeri positivi è di scartare il bit meno significativo. Poiché il Dos memorizza l'orario di creazione di un file con la precisione dei 2 secondi, la routine sarà in grado di confrontare le date e gli orari con la precisione dei 4 secondi: in altre parole, applicando la funzione a due file creati a meno di quattro secondi di distanza essa potrebbe fornire il medesimo risultato per entrambi. A parte questa piccola limitazione, la routine risolve appieno le

nostre esegenze. Notate che questa routine deve essere compilata con BASIC PDS o VBDOS, a causa della presenza dell'istruzione ON LOCAL ERROR; chi lavora con QuickBASIC deve eliminare l'istruzione in questione e controllare la presenza del file in un altro modo.

Ed ecco a voi il trucco ! Quando l'ambiente di sviluppo lancia il compilatore, attraverso una chiamata al Dos, possiamo fare in modo che sia invece richiamato un programma scritto da noi, che confronta la data del file sorgente con quella del file oggetto .OBJ corrispondente, ed effettua la compilazione solo se il file sorgente risulta effettivamente più recente del file OBJ, oppure se quest'ultimo non esiste affatto (cosa che accade la prima volta che si compila il programma). Ovviamente, poiché ci può essere un solo programma BC.EXE nella directory corrente, e questo deve essere il nostro programma, il "vero" compilatore deve trovarsi in un'altra directory, oppure deve essere stato rinominato in qualche altro modo, ad esempio "BCX.EXE".

Quando il Basic lancia il compilatore passa al Dos una stringa di comando di questo tipo:

```
BC C:\BASIC\TEST.BAS/O/Ot/Lr/FPi/T/C:512;
```

dove il punto e virgola finale indica al compilatore di non richiedere i parametri mancanti. Il seguente programma funge da "interfaccia" con il compilatore vero e proprio, che viene lanciato solo se il file sorgente è stato modificato dall'ultima compilazione, oppure se il file oggetto non esiste.

```
' Programma MYBC.BAS - prima versione
'
' Dopo aver compilato questo programma eseguire le seguenti operazioni
' dal prompt del Dos
'         ren bc.exe bcx.exe
'         ren mybc.exe bc.exe
'
' NOTA: non è riportato il listato della routine "FileDateTime"


'$INCLUDE: 'qbx.bi'
DEFINT A-Z

sourceFile$ = LEFT$(COMMAND$, INSTR(COMMAND$, "/") - 1)
objectFile$ = LEFT$(sourceFile$, INSTR(sourceFile$, ".")) + "OBJ"
sourceDate& = FileDateTime&(sourceFile$)
objectDate& = FileDateTime&(objectFile$)

' il confronto che segue funziona anche quando il file oggetto non
' esiste, e quindi objectDate& è uguale 1 -1
IF sourceDate& >= objectDate& THEN
        SHELL "BCX " + COMMAND$
END IF
```

I più accorti di voi si saranno probabilmente già accorti di un paio di difetti di questa prima versione del programma, e cioè :

- se il file sorgente non è stato modificato dall'ultima compilazione, ma esso contiene una o più istruzioni $INCLUDE che fanno riferimento a file

modificati, il programma erroneamente decide che non è necessario ricompilare il sorgente

- normalmente il BASIC controlla il livello di errore restituito dal compilatore, e termina di compilare i vari moduli del programma quando la compilazione di uno di essi produce errore; nel nostro caso, invece, il livello di errore restituito dal compilatore (bcx.exe) è perso

Fortunatamente la soluzione di entrambi i problemi non è troppo difficile: il primo si risolve eseguendo il parsing del programma sorgente alla ricerca di istruzioni di $INCLUDE e controllando la data dei vari file "inclusi"; il secondo analizzando l'output prodotto dal compilatore ed isolando la riga che riporta il numero degli errori prodotti. Il seguente listato riporta la nuova versione del programma che riporta questi miglioramenti; per non penalizzare troppo la velocità di esecuzione, il programma analizza solo le prime 50 righe del sorgente alla ricerca di istruzioni $INCLUDE; non è preso in considerazione il caso delle istruzioni di $INCLUDE nidificate (quando cioè il file di include fa riferimento ad un altro file di include, possibilità prevista dal BASIC ma raramente usata).

```
' Programma MYBC2.BAS - seconda versione
'
' Dopo aver compilato questo programma eseguire le seguenti operazioni
' dal prompt del Dos
'         ren bc.exe bcx.exe
'         ren mybc.exe bc.exe
'
'$INCLUDE: 'qbx.bi'
DEFINT A-Z

sourceFile$ = LEFT$(COMMAND$, INSTR(COMMAND$, "/") - 1)
objectFile$ = LEFT$(sourceFile$, INSTR(sourceFile$, ".")) + "OBJ"
sourceDate& = FileDateTime&(sourceFile$)
objectDate& = FileDateTime&(objectFile$)

' se il file sorgente è stato modificato oppure se il file oggetto
' non esiste il sorgente deve essere ricompilato
IF sourceDate& >= objectDate& THEN GOTO compila

' se il sorgente non è stato modificato dall'ultima compilazione
' occorre accertarsi che esso non dipenda da file di $INCLUDE che
' invece sono stati modificati - per non penalizzare troppo la velocità
' sono lette solo le prime 50 righe del sorgente

OPEN sourceFile$ FOR INPUT AS #1
linenum = 0

DO
        LINE INPUT #1, text$
        linenum = linenum + 1
        i = INSTR(text$, "$INCLUDE:")
        IF i <> 0 THEN
                firstChar = INSTR(i + 8, text$, "'") + 1
                lastChar = INSTR(firstChar, text$, "'") - 1
```

```
                includeFile$ = MID$(text$, firstChar, lastChar - firstChar +1)
                IF FileDateTime&(includeFile$) > sourceDate& THEN
                        CLOSE #1
                        GOTO compila
                END IF
        END IF
LOOP UNTIL EOF(1) OR linenum = 50
CLOSE #1

' se arriviamo ad eseguire questa riga non è necessario ricompilare
' il sorgente, per cui possiamo tornare al Basic senza segnalare errore
END

compila:
' lancia il compilatore, ridirezionando il suo output su un file
SHELL "BCX " + COMMAND$ + "> bcx.$$$"

' leggiamo l'output prodotto dal compilatore; la riga che ci interessa
' è l'ultima, che è del tipo "   XXX severe error(s)"
OPEN "bcx.$$$" FOR INPUT AS #2
DO
        LINE INPUT #2, text$
LOOP UNTIL EOF(2)
CLOSE #2
' se vi sono errori termina restituendo un errore al Basic
numErrori = VAL(text$)
IF numErrori <> 0 THEN END 1
' altrimenti termina normalmente...
```

Risolto il problema della compilazione possiamo concentrarci sul processo di linking; in questo caso non ci interessa confrontare le date dei file quanto avere la possibilità di intervenire sugli switch di link e di aggiungere uno o più stub file. Il meccanismo che useremo per intercettare la chiamata al linker è simile a quello esposto in precedenza a proposito del compilatore: rinomineremo il linker "vero" con un altro nome (LINKX.EXE) e creeremo un programma LINK.EXE "fasullo", il cui unico scopo è passare al linker il nome degli stub file. Nel caso del linker, il BASIC PDS passa i vari parametri attraverso un *response file*, usando una istruzione

```
LINK @~QBLNK.TMP
```

per aggiungere gli stub file occorre intervenire sul response file prima che il linker lo possa leggere. Una complicazione è data dal fatto che le varie componenti di questo file non sono separate dalla coppia Carriage Return + Line Feed ma da un semplice Line Feed, e quindi non possono essere separate leggendo il file con una istruzione LINE INPUT. In altri termini, l'intero file appare al BASIC composto da una sola riga di testo e può essere letto con un'unica istruzione LINE INPUT; sarà poi compito del nostro programma ricercare i Line Feed CHR$(10) per individuare il punto in cui inserire i nomi degli stub file; tale punto è in coda alla lista dei moduli oggetto che devono essere linkati, ed è individuato dalla prima "riga" (intesa nel senso appena descritto) che *non* termina con un simbolo "+". E' molto probabile che questa descrizione vi abbia confuso le idee, ma un attento studio del seguente listato e del manuale del BASIC dovrebbe risolvere tutti i dubbi.

```
' Programma MYLINK.BAS
'
' Dopo aver compilato questo programma eseguire le seguenti operazioni
' dal prompt del Dos
'       ren link.exe linkx.exe
'       ren mylink.exe link.exe
'

DEFINT A-Z
' la riga seguente indica quali switch e stub file devono essere   aggiunti
' in fase di link - in questo caso rimuoviamo il supporto per i file
' aperti su porta seriale, ed eliminiamo la descrizione per esteso dei
' messaggi di errore
switches$ = "/NOE"              ' /NOE è obbligatorio per usare gli stub
stubFiles$ = "+NOCOM +SMALLERR"

' ottieni il nome del response file, scartando il primo carattere "@"
' dal contenuto della riga di comando
responseFile$ = MID$(COMMAND$, 2)

' leggi il response file in un unica stringa
OPEN responseFile$ FOR INPUT AS #1
LINE INPUT #1, text$
CLOSE #1

' aggiungi in testa gli switch di link
text$ = switches$ + text$

' la variabile text$ contiene le varie righe del respose file, separate da
' caratteri di line feed (ASCII 10)- le istruzioni seguenti individuano la
' prima di tali righe che NON termina con un carattere "+", ed aggiunge a
' tale riga il nome degli stub file che ci interessano
index = 0
DO
        index = INSTR(index + 1, text$, CHR$(10))
        IF index = 0 THEN EXIT DO
        IF MID$(text$, index - 1, 1) <> "+" THEN
                text$ = LEFT$(text$, index - 1) + stubFiles$ + MID$(text$, _
                        index)
                EXIT DO
        END IF
LOOP

' possiamo riscrivere il response file ...
OPEN responseFile$ FOR OUTPUT AS #2
PRINT #2, text$
CLOSE #2

' ...ed eseguire infine il vero linker
SHELL "LINKX " + COMMAND$
```

Notate tra le prime istruzioni l'assegnazione alla variabile **switches$** delle opzioni di link (l'opzione /NOE è obbligatoria quando si usano gli stub file), e alla variabile **stubFiles$** dei nomi degli stub file che ci interessano; in un programma più raffinato potremmo prevedere di leggere questi valori da un file su disco o una variabile d'ambiente, in modo da non dover modificare e ricompilare il file MYLINK.BAS per ogni particolare progetto.

10

I BUG DEL MICROSOFT BASIC

L'interprete ed il compilatore BASIC sono programmi come tanti altri, creati da esseri umani come noialtri - magari un po' più in gamba - ma che comunque commettono errori. Purtroppo, se i nostri errori di programmazione influenzano soltanto i nostri prodotti e provocano al massimo le (giuste) rimostranze di qualche cliente, i *loro* errori si ripercuotono su migliaia (o forse milioni) di programmi creati in tutto il mondo usando questo linguaggio.

Se state pensando che per questo motivo il BASIC non sia adatto per la produzione di prodotti commerciali, preciso immediatamente che si tratta di problemi che affliggono indistintamente *tutti* i linguaggi di programmazione, inclusi quelli dell'ultima generazione (i vari linguaggi orientati agli oggetti, per intenderci). Se non altro il BASIC è uno dei linguaggi più collaudati, che vanta almeno una decina tra major e minor release, e che ha risolto molti dei problemi che infestano le prime versioni di qualsiasi software. In generale, il problema quindi non è di cercare un linguaggio di programmazione assolutamente privo di bug (che nella mia personale opinione non esiste), quanto quello di conoscere i limiti e i difetti del proprio linguaggio, imparando allo stesso tempo il modo di superarli.

Non tutte le versioni del BASIC sono afflitte dai medesimi bug; per questo motivo, dove ciò è stato possibile, ho aggiunto una annotazione che indica in quali versioni del linguaggio il problema è stato riscontrato, e quando e se è stato risolto in seguito. Noterete che ad ogni major release del linguaggio corrispondono - com'è prevedibile - il maggior numero di bug: questo è il caso del QuickBASIC 4.0 (il primo interprete a sfruttare la tecnologia p-code), che

fu infatti seguito a breve dalle due versioni minori 4.0a e 4.0b senza che ne sia mai stata data molta pubblicità; il QuickBASIC 4.5 oltre ad aggiungere poche nuove istruzioni, servì a risolvere molti dei problemi rimasti, ed è tuttora una versione abbastanza affidabile.

Il BASIC PDS 7.0 è un prodotto con relativamente pochi bug, e la release successiva 7.1 si rese necessaria quasi soltanto per rendere più efficiente il gestore ISAM. Il Visual BASIC per Dos deriva direttamente dal BASIC 7.1 e presenta quasi tutti i bug presenti in quest'ultima, ma essendo anche la prima versione *event-driven* aggiunge una serie di problemi nel funzionamento dei form.

Un'altra avvertenza: alle varie versioni di QuickBASIC sono corrisposte altrettante versioni del compilatore BASCOM (il file eseguibile BC.EXE, per intenderci), anche se i numeri di versione hanno cominciato a coincidere solo a partire dal BASIC Professional 7.0. Nelle note che seguono cercherò di specificare dove è possibile se essi si riferiscono solo all'interprete QB/QBX/VBDOS oppure al compilatore BC, oppure ad entrambi.

Infine, in alcuni casi il "bug" non si riferisce al software vero e proprio, bensì alla documentazione (carente o assente) oppure all'iterazione con il sistema operativo, i driver, la rete locale, ecc. Si tenga presente che in alcuni casi non mi è stato possibile duplicare l'errore, perché dipende da una particolare configurazione hardware/software a me non disponibile oppure percheé il bug sembra essere di natura intermittente; ad ogni modo ho deciso di includere una nota su tutti i bug di cui sono venuto a conoscenza, in modo da aiutare chi di voi dovesse eventualmente trovarsi in difficoltà.

ARRAY HUGE

Questo è un bug ben noto a chiunque abbia provato a creare array di record (variabili strutturate TYPE...END TYPE) per una occupazione di memoria complessiva maggiore di 128K; se la dimensione del record non è una potenza del due (ossia 4,8,16, ecc.) il BASIC non permette di superare tale limite. Facciamo subito un esempio:

```
TYPE myRecord
   nome AS STRING * 28
   autore AS STRING * 14
   id AS LONG
   copie AS INTEGER
END TYPE
```

La lunghezza di questo record è di 48 byte, per cui in condizioni normali (senza cioè che sia attivata l'opzione huge /AH) non è possibile creare un array di più di 1365 elementi (infatti 64K / 48 = 1365.333...); usando viceversa l'opzione /

AH il limite massimo è dato da 2730 elementi, e non può essere superato anche disponendo di centinaia di Kbyte di memoria libera. Il motivo di questo comportamento, presente in tutte le versioni del BASIC ma documentato solo con il BASIC 7.0, sta nel metodo di indirizzamento degli elementi dell'array.

Senza entrare troppo nei dettagli tecnici, quello che è più importante dal punto di vista strettamente pratico è trovare una soluzione a tale problema; io ve ne propongo un paio:

1. la prima soluzione, la più ovvia e immediata, è di aggiungere un elemento *dummy* ("fasullo") al record, in modo da portare la sua lunghezza complessiva alla più vicina potenza del 2; nel caso dell'esempio, si tratterebbe di aggiungere un elemento:

```
dummy AS STRING * 16
```

   in modo da portare la lunghezza del record a 64 byte; in tal caso le uniche limitazioni alla dimensioni dell'array saranno date dalla memoria disponibile (ad esempio 4196 elementi occupano 256K). L'ovvio svantaggio di questa tecnica è la memoria sprecata (ben 1/4 della memoria usata dall'array), soprattutto nel caso in cui la dimensione dell'elemento dummy supera i pochi byte

2. una tecnica alternativa è quella di "spezzare" il record in due (o più) strutture distinte, e fare in modo che le due strutture siano lunghe esattamente quanto una potenza del due; nel nostro caso potremmo creare due strutture:

```
TYPE myRecord1
   nome AS STRING * 28
   id AS LONG
END TYPE
TYPE myRecord2
   autore AS STRING * 14
   copie AS INTEGER
END TYPE
```

   che sono lunghe rispettivamente 32 e 16 byte, e non rientrano quindi nella limitazione enunciata in precedenza. Lo svantaggio di questa soluzione è che richiede una riscrittura del programma abbastanza profonda, ma spesso si tratta dell'unica soluzione effettivamente praticabile

Si noti che questo bug riguarda esclusivamente gli array di record, in quanto tutti i tipi predefiniti (integer, long, single, double e currency) sono lunghi 2, 4 o 8 byte e sono quindi sempre potenze del due.

LE ISTRUZIONI INT, CINT E CLNG

Ecco un bug davvero anomalo: provate ad eseguire (in modo interpretato o compilato) le seguenti righe di programma:

```
PRINT INT(.9 * 10)  ' stampa 8 (sbagliato)
PRINT .9 * 10  ' stampa 9 (corretto)
a! = .9 * 10
PRINT a!   ' stampa 9 (corretto)
```

il bug è determinato dal fatto che la precisione finita del calcolatore in virgola mobile calcola per la moltiplicazione un risultato leggermente inferiore al valore 9, che però viene correttamente arrotondato dalla routine di stampa; se invece tale valore è prima inviato alla routine INT essa ne calcola (a suo modo correttamente) la parte intera. Come mostra il precedente esempio, una possibile soluzione è quella di assegnare il risultato dell'espressione ad una variabile temporanea, in modo che l'arrotondamento al valore corretto avvenga prima del calcolo della parte intera.

Le istruzioni CINT e CLNG arrotondano un numero all'intero più vicino; quello che la Microsoft non documenta nei manuali del linguaggio è che quando la parte decimale è pari esattamente a 0.5 la funzione di comporta differentemente a seconda che la parte intera sia pari o dispari: se la parte intera è dispari CINT arrotonda verso l'alto, se invece la parte intera è pari arrotonda verso il basso. Proverò ad illustrare il concetto con un esempio:

```
PRINT CINT(2.5) ' il risultato è 2
PRINT CINT(3.5) ' il risultato è 4
```

Per quanto può risultare strano, questo comportamento è voluto (in altre parole: non è un bug), in quanto permette di costruire corretti programmi di statistica, ed è probabilmente condiviso da altri linguaggi di programmazione Microsoft.

SUBROUTINE SENZA ARGOMENTI

Si tratta di un bug che si riscontra nell'ambiente integrato QB/QBX/VBDOS e che crea un bel po' di grattacapi fino a quando non si decide di usare il debugger per controllare il flusso dell'esecuzione. Si consideri la seguente riga di programma

```
MyProcedure: CLS
```

dove **MyProcedure** è una subroutine che non richiede argomenti, regolarmente dichiarata con una DECLARE e definita altrove nel programma. Per quanto può sembrare assurdo, il BASIC *non esegue la procedura, come se non esistesse* ! Il bug è evidentemente causato dal fatto che il parser considera la chiamata alla procedura seguita dal simbolo dei due punti come se fosse una etichetta.

A questo punto è facile rimediare, usando l'istruzione CALL oppure scrivendo i comandi su due righe differenti. È incredibile come, pur conoscendo bene il problema, io stesso continui regolarmente a cascarci ogni volta! Si tratta di un bug presente in tutte le versioni del BASIC; in Visual BASIC per Dos lo stesso problema si presenta anche per i metodi senza argomenti, come in:

```
Form1.PRINTFORM: CLS
```

QUICK LIBRARY IN QB 4.0

Si tratta di un problema abbastanza evidente, che si manifesta quando si compila dall'interno del QB 4.0 un programma che utilizza una Quick Library; per come è costruita la stringa di comando passata al linker, l'effetto è che l'intera libreria LIB corrispondente è linkata al programma, anziché le sole routine effettivamente richiamate dal programma stesso. Di conseguenza le dimensioni dell'eseguibile crescono a dismisura; l'unica soluzione possibile è di compilare dalla linea di comando. Questo bug è stato corretto nelle versioni successive, ma ho voluto riportarlo anche per mostrare quanto sia facile anche per una grande software house rilasciare un prodotto con alcuni bug davvero grossolani (possibile che nessuno in Microsoft si sia accorto di una cosa del genere durante la fase di test ?).

CARATTERI ASCII NON COMPATIBILI

I due caratteri con codice ASCII pari a 1 e 2 non dovrebbero essere inseriti come stringhe in un programma BASIC, in quanto hanno la sgradevole prerogativa di confondere le idee al compilatore. Accade cioè che un programma che contiene questi caratteri in una stringa o in una istruzione DATA sia eseguito correttamente con l'interprete, ma non possa essere compilato con BC.EXE o peggio compili correttamente ma produca errore a runtime. La soluzione è ovviamente quella di usare le funzione CHR$(1) e CHR$(2).

Un problema analogo si riscontra in QuickBASIC 4.5 con il carattere ASCII 128 "Ç": se un programma contiene questo carattere e viene salvato nel formato binario, il compilatore non riuscirà a compilarlo. Anche in questo caso possiamo utilizzare la funzione CHR$, o più semplicemente salvare il file in formato ASCII; il problema non si riscontra nelle altre versioni del BASIC.

Giacché siamo in argomento, non tutti sanno come introdurre caratteri di controllo (con codice ASCII compreso tra 1 e 31) direttamente nel sorgente del programma. Il metodo è infatti sepolto tra le pagine del manuale: occorre

premere la combinazione CTRL-P (apparirà l'indicazione "^P" in basso sulla riga di tasto), facendola seguire dal carattere di controllo desiderato, che può anche essere composto con il tastierino numerico usando la solita procedura ALT+codice numerico. Ad esempio, per comporre il carattere ASCII 3 (^C) occorre premere CTRL-P seguito da CTRL-C, per comporre il carattere ASCII 30 occorre invece premere CTRL-P e poi comporre il codice 030 sul tastierino numerico mentre si tiene premuto il tasto ALT.

L'ISTRUZIONE SWAP

L'istruzione SWAP è spesso trascurata, a torto in quanto spesso essa è più efficiente di una serie di assegnazioni. Per evitare problemi, però, occorre essere consci di alcuni malfunzionamenti dell'istruzione SWAP:

1. se si tenta di usare SWAP per scambiare tra loro due componenti di un record, l'interprete può causare un crash di sistema durante la fase di "binding" (generazione del p-code); il problema non si manifesta sotto il compilatore, né quando si applica SWAP ad interi record
2. in QuickBASIC 4.5 l'interprete può provocare un crash quando si tenta di eseguire lo swap tra due stringhe, di cui una è una costante; le altre versioni del BASIC segnalano correttamente l'errore in fase di binding
3. durante la compilazione può avvenire un "Internal error" se si tenta di eseguire lo swap tra due elementi numerici di un record in una subroutine ed è stata precedentemente usata la funzione VAL per assegnare un valore ad uno o entrambi gli elementi del record. Ecco un esempio pratico:

```
SUB testSub (res AS resType)
 res.x = VAL(a$)
 res.y = VAL(b$)
 SWAP res.x, res.y
END SUB
```

Vi sono due possibili soluzioni al problema: (a) assegnare il risultato di VAL ad una variabile temporanea, e poi assegnare il contenuto di tale variabile al componente del record, (b) compilare il programma con l'opzione /X.

LE FUNZIONI DEF FN

Anche se tendono ad essere soppiantate dalle più strutturate FUNCTION, che hanno il vantaggio di poter essere situate anche in moduli differenti, le DEF FN "vecchia maniera" continuano a essere utili, soprattutto perché la loro

esecuzione è sensibilmente più veloce delle FUNCTION corrispondenti. Provate a lanciare (dopo averlo compilato) il seguente programma:

```
DEF FNMaxInt% (x%, y%)
 IF x% > y% THEN FNMaxInt% = x% ELSE FNMaxInt% = y%
END DEF
t! = TIMER
FOR i% = 1 TO 10000: a% = FNMaxInt%(i%, 100): NEXT
PRINT "Tempo impiegato da DEF FN"; TIMER - t!
t! = TIMER
FOR i% = 1 TO 10000: a% = MaxInt%(i%, 100): NEXT
PRINT "Tempo impiegato da FUNCTION"; TIMER - t!
FUNCTION MaxInt% (x%, y%)
 IF x% > y% THEN MaxInt% = x% ELSE MaxInt% = y%
END FUNCTION
```

Il motivo di questo aumento di velocità è dovuto al fatto che le chiamate alla routine FN sono ottenute mediante NEAR CALL (chiamate intra-segmento), mentre le FUNCTION sono sempre richiamate con FAR CALL (chiamate inter-segmento, notoriamente più lente), anche se si trovano nella stesso modulo dell'istruzione chiamante. L'effetto è che, se il corpo della funzione è molto breve (come nell'esempio), il tempo speso a richiamare la funzione diventa una frazione non trascurabile del tempo complessivo di esecuzione. Tuttavia, c'è un problema con DEF FN che non può essere trascurato. Gli argomenti passati ad una funzione FN sono implicitamente STATIC, per cui tutta la memoria allocata a questi ultimi rimane inutilizzata fino alla chiamata successiva. Un esempio chiarirà le idee:

```
DEF FNCapitalize$ (a$)
 FNCapitalize$ = UCASE$(LEFT$(a$, 1)) +
LCASE$(MID$(a$, 2))
END DEF
b$ = STRING$(1024, "a")
PRINT FNCapitalize$(b$)
```

Il programma passa alla funzione **FNCapitalize$** un argomento lungo esattamente 1K, attraverso l'argomento **a$**; questa memoria in DGROUP non è rilasciata fino alla successiva chiamata alla funzione, ma nel frattempo il programma BASIC avrà meno spazio a disposizione per le proprie stringhe, le variabili semplici e gli array statici; questo problema si manifesta con tutte le versioni del compilatore e dell'interprete. Se la funzione non ha alcun effetto collaterale (ad esempio la modifica di altre variabili globali), la memoria può essere rilasciata semplicemente richiamando la funzione con un argomento nullo:

```
temp$ = FNCapitalize$("")
```

LO STACK

Nella maggior parte dei programmi la dimensione di default dello stack (circa 3K per default) è sufficiente e non vi è la necessità di modificarla. Le cose

cambiano quando il programma usa pesantemente la ricorsione (nel qual caso essa deve essere aumentata) oppure ha bisogno di ulteriore spazio in DGROUP (nel qual caso può essere necessario diminuirla). E' importante sapere che nei programmi interpretati uno stack insufficiente è segnalato con un errore di "stack overflow", mentre nei programmi compilati la medesima situazione genera direttamente un crash di sistema, senza alcun messaggio di errore. Se volete tenere sotto controllo lo stack in un programma compilato potete provare a compilarlo con l'opzione /D, che inoltre si preoccupa di verificare che gli indici nelle operazioni con gli array sono sempre nei limiti consentiti. Come è noto, la dimensione dello stack si imposta con le istruzioni CLEAR o STACK:

```
CLEAR , , newStackSize&  ' funziona in tutte le versioni
STACK newStackSize&  ' funziona solo in BASIC PDS e VBDOS
```

La differenza tra le due è che CLEAR azzera tutte le variabili.

Sempre a proposito dello stack, è importante sapere che a differenza di quanto descritto nella documentazione, l'istruzione CHAIN resetta lo stack alla dimensione di default. Il problema è tanto più grave nelle versioni fino a QB 4.5, in cui non è possibile usare CLEAR nel modulo richiamato mediante CHAIN, poiché tale istruzione azzererebbe le variabili e in definitiva annullerebbe tutti i vantaggi dell'operazione di CHAIning. L'unica soluzione valida con tutte le versioni del BASIC è di usare l'opzione /ST:xxxx del linker per impostare una volta la dimensione dello stack in fase di compilazione.

L'ISTRUZIONE INCLUDE

In senso stretto l'istruzione INCLUDE può essere considerata superflua, in quanto l'operazione di inclusione di un file potrebbe essere benissimo sostituita da una operazione di taglia-e-incolla dell'editor; né d'altra parte essa permette di risparmiare memoria, in quanto - diversamente da quanto si può essere portati a credere - le istruzioni "incluse" nel programma occupano la medesima quantità di memoria che occuperebbero se fossero inserite nel sorgente: per rendersene conto basta premere (nell'ambiente QBX o VBDOS) il tasto F2 ed ottenere la lista delle procedure dichiarate insieme alla loro occupazione di memoria in Kbyte. Nella pratica giornaliera l'istruzione di INCLUDE è comunque utilissima per inserire rapidamente le istruzioni DECLARE delle funzioni definite in una Quick Library, oppure per inserire un numero di funzioni DEFFN già scritte e collaudate in altri programmi; non è invece possibile "includere" procedure SUB o FUNCTION, che devono essere forzatamente caricate come moduli separati.

Nella preparazione di programmi multi-modulo, i file di INCLUDE sono praticamente indispensabili, come unico mezzo per accertarsi che le variabili

COMMON siano dichiarate nel medesimo ordine in tutti i moduli (se non lo fossero il programma non funzionerà mai come dovrebbe) e che le DECLARE delle procedure BASIC siano consistenti con la loro effettiva implementazione. Abituati come siamo alla comodità dell'ambiente integrato, ci scordiamo spesso di una "mancanza" del BASIC (chiamarlo bug è troppo, anche perché è condiviso da altri linguaggi), e cioè che quando si compila il programma manualmente il compilatore vede un modulo per volta, e non può quindi accertarsi che la DECLARE di una procedura BASIC presente in un altro modulo sia corretta per numero e tipo di argomenti. Ecco un esempio:

```
' modulo UNO.BAS
DECLARE SUB testSub (primo%, secondo%)
DECLARE FUNCTION testFun% (primo%)
testSub 1, 4
result% = testFun% (7)
' modulo DUE.BAS
SUB testSub (primo%, secondo#)
...
END SUB
' modulo TRE.BAS
FUNCTION testFun% (primo%, secondo$)
...
END FUNCTION
```

compilando i tre moduli dal prompt del Dos o da un file batch, non viene segnalato alcun errore sintattico, ma il programma è sicuramente errato in quanto: (a) la procedura **testSub** attende come secondo argomento un valore DOUBLE, mentre la chiamata passa un valore INTEGER e non viene eseguita alcuna conversione automatica; (b) se la procedura **testSub** modifica il valore di **secondo#** essa scriverà in una zona di memoria occupata da altre variabili, con risultati facilmente immaginabili, (c) la funzione **testFun** attende due parametri, ma il programma principale la chiama con un solo argomento: non esiste metodo migliore per assicurarsi un crash del sistema !

Raggruppando tutte le DECLARE in un unico file ed inserendo delle opportune INCLUDE in tutti i moduli del programma, si scongiura questo pericolo: infatti, la dichiarazione della procedura **testSub** sarà inclusa anche nel modulo DUE.BAS, per cui il compilatore segnalerà errore e permetterà di porre rimedio.

Stabilita quindi l'importanza dell'istruzione di INCLUDE nella scrittura di programmi corretti, mi affretto ad indicare un paio di piccoli bug che riguardano l'istruzione medesima:

1. se l'ultima linea del file di INCLUDE non termina con un carattere di carriage return, possono accadere strane cose al programma, e molti errori inspiegabili; in molti casi questi possono essere evitati aggiungendo nel programma principale una riga vuota dopo l'istruzione INCLUDE, ma la cosa migliore rimane quella di controllare che il file di include termini regolarmente (il problema si pone soprattutto quando

il file è creato usando un word processor o comunque un editor diverso da quello integrato nell'ambiente)

2. è buona norma inserire l'istruzione INCLUDE prima di scrivere il resto del programma; accade infatti che se il file di include contiene una istruzione DEFINT (o DEFLNG, DEFDBL, ecc.) essa sarà riportata in tutte le procedure e funzioni create da quel momento in poi, ma non sarà aggiunta a quelle già esistenti - questa considerazione vale ovviamente anche per le istruzioni DEFINT inserite manualmente nel programma al di fuori di un file di include.

INTERI LONG IN QUICKBASIC 4.X

Vi sono numerosi bug che riguardano il trattamento degli interi a 32 bit in QuickBASIC 4 e 4.5; fortunatamente questi sono stati corretti nelle versioni più recenti, ma le riporto comunque a beneficio di coloro - e non sono pochi - che continuano ad usare vecchie versioni del linguaggio.

1. se una struttura (record) contiene elementi LONG ed essi sono usati nei calcoli in una procedura i risultati non sono affidabili nei programmi compilati; ad esempio:

```
TYPE esempioType
 numero AS LONG
END TYPE
CALL testSub
SUB testSub
DIM array(100) AS esempioType
array(1).numero = 4
array(2).numero = 6
PRINT array(1).numero * array(2).numero
END SUB
```

il risultato cambia a seconda che si usi l'opzione /O o meno; questo bug non riguarda l'interprete, nel quale i programmi sono sempre eseguiti con l'opzione di debug attivata, per cui si tratta di un errore particolarmente pericoloso in quanto influenza programmi compilati che, avendo superato la fase di test nell'ambiente integrato, sono erroneamente considerati completamente "testati".

2. errori simili al precedente si manifestano - ma soltanto nei programmi compilati - quando si passa un vettore di LONG come argomento ad una procedura e questa li utilizza nei calcoli. Un possibile rimedio sembra essere quello di evitare di usare gli elementi dell'array nelle espressioni, utilizzando invece delle variabili LONG temporanee a cui assegnare gli elementi prima di effettuare i calcoli.

3. le matrici bidimensionali di interi LONG sono quelle con maggiori problemi: se si dimensiona una matrice di questo tipo il BASIC non riesce ad accedere correttamente agli elementi di indice zero, a meno che il sistema non disponga di un coprocessore matematico (incredibile!). Questo comportamento riguarda solo i programmi compilati senza l'opzione /D (debug), mentre l'interprete non ne risente in quanto i programmi sono sempre eseguiti con l'opzione di debug attivata.

CALCOLI SUGLI ELEMENTI DI UN RECORD

In alcuni casi il compilatore BC può produrre un errore "Out of Memory" se il programma contiene delle procedure SUB a cui è passato un record TYPE come argomento, e la procedura esegue dei calcoli su una o più componenti del record; questo problema riguarda il BASIC PDS e il VBDOS, ed è causato dal tentativo (evidentemente maldestro) del compilatore di ottimizzare la sequenza dei calcoli.

Esistono diverse soluzioni al problema: (a) in alcuni casi è sufficiente modificare la struttura del record, cambiando l'ordine con cui le componenti compaiono nella definizione TYPE, (b) la variabile record può essere rimossa dalla lista degli argomenti e resa accessibile alla procedura con una istruzione DIM SHARED o COMMON SHARED, oppure (c) il programma può essere compilato con l'opzione /X, che tra gli altri effetti ha quello di inibire alcune delle ottimizzazioni tentate dal compilatore.

L'ISTRUZIONE ENVIRON E LA FUNZIONE ENVIRON$

Pur non essendo veri è propri bug, ambedue queste istruzioni sono soggette a problemi, che si risolvono solo dopo aver ben spiegato alcuni particolari su come e dove il Dos memorizza le proprie variabili d'ambiente. Tanto per incominciare, è molto probabile che una istruzione ENVIRON come la seguente:

```
ENVIRON "QBPATH=c:\vbdos"
```

provochi un errore "Memoria insufficiente". Ecco il motivo: il sistema operativo assegna a ciascun programma un environment distinto, per cui possiamo parlare di un *environment master* (relativo al COMMAND.COM principale) e di uno o più *environment secondari* (relativi ai COMMAND.COM secondari e a tutti i programmi applicativi). Una differenza importante tra i due tipi di enviroment è che la dimensione del primo è fissata al bootstrap (dal parametro /E della

direttiva SHELL in CONFIG.SYS) ed è occupato solo parzialmente dalle variabili d'ambiente definite in seguito, ad es. COMSPEC, PATH, PROMPT e le altre variabili definite in AUTOEXEC.BAT; viceversa, la dimensione degli environment secondari è pari esattamente al numero di byte occupati dalle variabili nell'environment master al momento della loro creazione. Se da una parte questa strategia permette di contenere lo spreco di memoria, dall'altra causa problemi come quelli visti poc'anzi: l'istruzione ENVIRON fallisce perché non vi è spazio libero in memoria.

Svelato il mistero è abbastanza semplice trovare la risoluzione; se il vostro programma deve creare una nuova variabile d'ambiente (oppure assegnare ad una variabile esiste una stringa più lunga del suo corrente valore) occorre creare una variabile temporanea fasulla *prima* di lanciare il programma stesso, ad es:

```
SET TMP=@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
myprog
```

in tal modo la variabile TMP sarà creata nel master environment e quindi ereditata nell'environment secondario. Una volta partito il programma BASIC, una delle prime sue istruzioni dovrebbe essere:

```
ENVIRON "TMP="
```

in modo da liberare un po' di memoria nell'environment secondario e fare funzionare regolarmente tutte le istruzioni ENVIRON successive. Attenzione ad assegnare alla variabile temporanea un numero di caratteri almeno pari al numero di byte richiesti da tutte le istruzioni ENVIRON impiegate dal programma.

Il meccanismo adottato dal Dos per creare i vari environment secondari spiega anche un'altro strano comportamento dell'istruzione ENVIRON che da molti è ingiustamente interpretato come un bug del BASIC. Supponiamo di eseguire da BASIC l'istruzione:

```
' aggiungi la directory corrente al path di sistema
ENVIRON "PATH=" + ENVIRON$("PATH") + CURDIR$
```

al termine del programma eseguiamo un comando SET dal prompt del Dos e scopriamo che il path di sistema non è stato affatto alterato come volevamo: è accaduto che il Dos, al termine dell'esecuzione del programma, ha scartato l'environment secondario associato a quest'ultimo ed è tornato ad attivare il master environment. Poiché, come abbiamo detto, l'istruzione ENVIRON agisce sull'environment locale, il suo effetto scompare quando questo è scartato dal Dos. Ma allora, a che serve l'istruzione ENVIRON se il suo effetto è perso uscendo dal programma ? Non a molto, a dire il vero, e l'unica sua applicazione è di preparare un environment opportuno ai programmi lanciati dal BASIC con una istruzione SHELL. Ecco un esempio:

```
' aggiungi la directory D:\MYDIR al path di sistema
ENVIRON "PATH=" + ENVIRON$("PATH") + "D:\MYDIR"
```

```
' mostra che il nuovo path è stato attivato
 SHELL  "SET"
' lancia un programma applicativo tramite un command.com
' secondario; il programma sarà eseguito anche se si
' trova nella directory D:\MYDIR che ora appartiene al path
 SHELL  "testprg"
```

Un'altra applicazione interessante di ENVIRON consiste nel modificare il prompt di sistema in modo da mostrare continuamente un messaggio all'utente per informarlo su come rientrare nel programma BASIC da una uscita temporanea al Dos:

```
ENVIRON "PROMPT=Batti EXIT+Invio per rientrare nel programma $_$p$g"
```

Notate la sequenza "$_" che inserisce una andata a capo nel prompt, e produce un messaggio di due righe:

```
Batti EXIT+Invio per rientrare nel programma
C:\QBASIC>
```

Un'ultima importante accortezza nell'uso della istruzione ENVIRON è di riportare il nome della variabile in maiuscolo, *esattamente come appare nell'output del comando* SET del Dos. Se ad esempio provate ad eseguire un comando:

```
ENVIRON "Prompt=Orario $t  $p$g"
```

otterrete il risultato di creare (spazio permettendo) una nuova variabile d'ambiente di nome "Prompt", lasciando indisturbata la variabile "PROMPT", che è poi quella usata dal sistema operativo per i suoi scopi. Per convincervi, provate ad eseguire una istruzione SHELL "SET" senza uscire dal programma. E' evidente che il Dos distingue tra nomi in maiuscolo e in minuscolo, ma converte automaticamente in maiuscolo i riferimenti alle variabili d'ambiente nei file batch o nelle istruzioni SET, per cui ad esempio le istruzioni:

```
SET path=c:\dos
SET PATH=c:\dos
```

sono equivalenti, come pure lo sono:

```
ECHO %path%
ECHO %PATH%
```

Per quanto detto finora, creando con ENVIRON una variabile d'ambiente contenente una o più lettere minuscole possiamo essere certi che essa non potrà essere modificata mediante il comando SET standard; i programmatori della stessa Microsoft hanno utilizzato questa tecnica nelle sessioni Dos eseguite sotto Windows 3.x, nel cui environment appare una variabile d'ambiente chiamata "windir" (in minuscolo) che contiene il percorso della directory principale di Windows stesso (si provi ad impartire un comando SET in una finestra Dos sotto Windows).

Quanto detto per il comando ENVIRON vale anche per la funzione ENVIRON$, che richiede che il nome della variabile sia specificato *esattamente come è riportato nell'environment*, rispettando cioè le maiuscole e le minuscole. Questo significa che le due istruzioni seguenti:

```
PRINT ENVIRON$ ("PATH")
PRINT ENVIRON$ ("path")
```

non sono affatto equivalenti, e soltanto la prima di esse restituisce il valore atteso. Tornando a quanto detto qualche riga sopra, questo comportamento ci fornisce un sistema quasi infallibile per determinare se il nostro programma è eseguito da Windows o meno:

```
IF ENVIRON$("windir") <> "" THEN
 PRINT "Eseguito da Dos"
ELSE
 PRINT "Eseguito da Windows 3.x"
END IF
```

Prima di terminare con la funzione ENVIRON$, vorrei far notare che è anche possibile passare un valore intero come argomento, nel qual caso la funzione restituisce la N-esima variabile di ambiente, nel formato "nome=valore". In tal modo possiamo scandire il contenuto dell'environment incrementando opportunamente il valore del parametro:

```
n% = 1
DO
 PRINT ENVIRON$(n%)
 n% = n% + 1
LOOP WHILE ENVIRON$(n%) <> ""
```

Queste righe di codice hanno lo stesso effetto di una istruzione SHELL "SET", ma sono più veloci in quanto non richiedono il caricamento di un processore dei comandi secondario.

IL COPROCESSORE MATEMATICO

Ecco alcuni report su dei bug che hanno a che fare con la presenza del coprocessore matematico; in alcuni casi, non disponendo dell'hardware e software descritto, non ho potuto effettuare le necessarie verifiche e mi limito a riportare queste informazioni come le ho trovate io stesso:

- il compilatore BASCOM 7 è il primo che testa la presenza del coprocessore matematico verificando se uno switch sulla scheda madre è attivato o meno; tutto file liscio, se non fosse che alcune schede madre - tra cui la Intel Inboard/PC - funzionano bene solo con tale switch disattivato, per cui su tali sistemi i programmi compilati con questa versione del compilatore non riconosceranno il coprocessore matematico
- i programmi compilati con BASCOM 6.00 e 6.00b che usano la libreria di default /FPI possono generare inspiegabili errori di overflow quando sono eseguiti su sistemi 286 o superiori con coprocessore matematico e con Dos 3.2 e 3.3; una possibile soluzione sembra essere quella di usare la libreria /FPI alternativa

- infine una combinazione hardware/software che sembra fatta apposta per dare grattacapi: se un programma è compilato con /FPI, usa i servizi di BTrieve (versioni 4.x e 5.x) ed esegue delle istruzioni CHAIN esso provocherà un errore "Illegal function call" su un sistema dotato di coprocessore matematico, a meno di non eseguire un "reset" di Btrieve prima dell'operazione di CHAIN.

PROGRAMMI TROPPO GRANDI

Il BASIC ha purtroppo la tendenza a comportarsi in modi strani quando il programma diventa troppo grande, in termini di numero di variabili e di quantità di moduli e procedure, e ho persino letto (ma non sperimentato personalmente) di errori di tipo "Duplicate definition" quando si tenta di usare variabili che si chiamano F$ oppure F% !

Molte delle mie esperienze al riguardo risalgono a qualche anno fa, e si riferiscono al QuickBASIC 4.5 e al BASIC PDS 7.0; stavo sviluppando un programma che era cresciuto a dismisura e aveva superato il limite della 15000 istruzioni, quando incominciai a sperimentare bug a dir poco inquietanti. Molte routine che avevano funzionato perfettamente fino a quel momento cominciarono a fare le bizze, fornendo risultati completamente errati. Peggio ancora, in alcuni casi il problema si manifestava sono nei programmi compilati, e non in quelli interpretati, per cui il debug era ancora più difficile.

Alla fine, lavorando con il CodeView e disseminando il programma di PRINT, riuscii a trovare che in alcuni casi i calcoli che le routine effettuavano sui valori passati come argomenti nelle CALL erano errati, come se il valore usato nelle espressioni non fosse quello corretto. Evidentemente il programma aveva delle richieste di memoria tali che il gestore della memoria del BASIC incorreva qualche piccolo errore, che però aveva conseguenze disastrose sul programma. Per fortuna la soluzione non era troppo difficile, e fu sufficiente assegnare il valore dell'argomento ad una variabile temporanea e usare la variabile nelle espressioni, oppure di evitare del tutto il passaggio di argomenti e comunicare con le varie procedure per mezzo di variabili globali. Morale della storia: se scrivete programmi molto lunghi, e ad un certo punto cominciano a manifestarsi strani errori nei calcoli, controllate nuovamente le procedure in questione, e non date per scontato che l'errore non può trovarsi in quel punto del programma solo perchè è stato controllato tante altre volte e ha sempre funzionato.

Un altro problema collegato ai programmi di grosse dimensioni è il cosiddetto *variable aliasing*, così denominato dalla stessa Microsoft, che si manifesta quando una procedura accede alla stessa locazione di memoria (cioè alla

stessa variabile) in due modi diversi, ad esempio attraverso un parametro e direttamente con il nome della variabile (se quest'ultima è SHARED) oppure se la variabile è passata alla routine due volte attraverso argomenti distinti. Facciamo un esempio:

```
numero = 4
PRINT SommaQuadrati (numero, numero)
...
FUNCTION SommaQuadrati (arg1, arg2)
 arg1 = arg1 * arg1
 arg2 = arg2 * arg2
 SommaQuadrati = arg1 + arg2
END FUNCTION
```

A prima vista sembrerebbe che questo frammento di programma dovrebbe mostrare il valore 32 (ossia 4^2 + 4^2), mentre in realtà il risultato è 512; il meccanismo diventa chiaro quando si nota che la funzione *modifica* il valore del parametro **arg1**, che essendo stato passato per riferimento (cioè per indirizzo), punta in realtà alla stessa locazione di memoria a cui punta **arg2**, il quale quindi viene anch'esso modificato, anche se questo particolare non è evidente se si limita l'analisi alle sole righe comprese nel corpo della FUNCTION. Accade quindi che:

```
arg1 = arg1 * arg1
```

calcola il quadrato dell'argomento (4 * 4 = 16) e lo memorizza nella locazione di memoria a cui punta sia **numero** che **arg1** che **arg2**; subito dopo l'assegnazione:

```
arg2 = arg2 * arg2
```

assegna al parametro **arg2** il valore 256 (cioè 16*16), ma per quanto detto sopra ne sono influenzate anche la variabile **numero** e il parametro **arg1**; infine, l'assegnazione:

```
SommaQuadrati = arg1 + arg2
```

restituisce al programma principale il valore 512 (256+256), palesemente errato. Inoltre, un altro effetto indesiderato, la variabile **numero** contiene ora il valore 256, e questo può portare ad ulteriori errori logici nel programma.

E' tutto chiaro quindi, e si tratta di un problema legato al passaggio di argomenti per indirizzo anziché per valore; a dimostrazione di ciò si noti che sostituendo nel programma la chiamata alla procedura con una delle due linee seguenti:

```
PRINT SommaQuadrati (4, 4)
PRINT SommaQuadrati ((numero), (numero))
```

tutto funziona alla perfezione; notate infatti che nella seconda riga le parentesi che racchiudono i parametri forzano il passaggio per valore anziché per indirizzo; se poi lavorate con BASIC PDS 7.1 o Visual BASIC potete anche esplicitare che il passaggio deve avvenire per valore usando la parola BYVAL nel definizione della FUNCTION:

```
FUNCTION SommaQuadrati (BYVAL arg1, BYVAL arg2)
```

Quello che invece resta oscuro è il motivo per cui la Microsoft indica nei manuali (ad esempio, a pag. 64 della *Programming Guide* del BASIC PDS 7.1) che questo problema si manifesta solo nei programmi di grosse dimensioni; mentre abbiamo appena visto che, lungi dall'essere un bug. si tratta di comportamento perfettamente prevedibile e comune a tutti i linguaggi di programmazione che per default passano gli argomenti per indirizzo anziché per valore, e che per giunta si manifesta qualunque sia la dimensione dei programmi.

CALCOLI SU INTERI E ARRAY NEI PROGRAMMI COMPILATI

Quando eseguiamo un programma nell'ambiente di sviluppo, il BASIC controlla tutti i calcoli e correttamente genera un "Overflow error" quando il risultato di una operazione tra INTEGER o LONG supera rispettivamente i 16 e i 32 bit di precisione. Ad esempio, nell'ambiente QB/QBX/VBDOS la seguente istruzione genera errore:

```
result = 1000 * 900
```

in quanto il risultato non può essere memorizzato in un intero a 16 bit; se questo è abbastanza chiaro, può invece stupire che continuiamo ad ottenere errore anche utilizzandouna variabile LONG, come in:

```
result& = 1000 * 900
```

Il motivo è semplice: quando il BASIC calcola il prodotto richiama la routine che moltiplica due INTEGER tra loro, per cui è il risultato intermedio a provocare l'errore, non l'operazione di assegnazione alla variabile. Compreso il problema la soluzione è abbastanza semplice, ed è sufficiente forzare il BASIC ad usare la routine che moltiplica tra loro due interi LONG, ad esempio:

```
result& = CLNG(1000) * 900
```

oppure:

```
result& = 1000& * 900
```

In definitiva, quindi, non si tratta di un vero bug, ma un compromesso che dobbiamo accettare tra la facilità di programmazione e il tentativo di ottimizzare l'esecuzione fatto dal BASIC. Questo bug, infatti, non si manifestava con GW-BASIC, che usava per default i valori a singola precisione.

Ma le cose non sono così semplici, perché tutto quello che ho detto è vero esclusivamente all'interno dell'ambiente di sviluppo: nei programmi compilati le righe viste sopra non producono alcun tipo di errore, ma assegnano alla variabile un risultato errato; più precisamente, accade che il codice prodotto dal compilatore calcola correttamente il prodotto usando una istruzione Assembly IMUL a 32 bit, ma poi scarta in ogni caso la word più significativa del risultato, senza testare se è diversa da zero. Il problema non si manifesta mai

nelle espressioni su interi LONG o su valori in virgola mobile: in questo caso infatti i calcoli sono eseguiti mediante una chiamata a subroutine, che provvede sempre a controllare la validità del risultato.

Un tipo di bug di natura completamente differente ma ricollegabile al precedente è l'indirizzamento ad elementi non esistenti di vettori e matrici; ad esempio, le seguenti istruzioni:

```
DIM a%(1000)
PRINT a%(1001)
```

provocano correttamente errore se eseguite nell'ambiente di sviluppo, mentre procedono indisturbate quando sono compilate in un programma standalone. Nei programmi interpretatati gli accessi ai singoli elementi di vettori e matrici avvengono attraverso una chiamata a subroutine, che tra gli altri compiti controlla che l'indice sia nell'intervallo consentito; al contrario, per ragioni di efficienza, nei programmi compilati l'accesso avviene direttamente alla memoria per mezzo di puntatori, mediante una sequenza di istruzioni Assembly simile alla seguente:

```
MOV BX,1001  ' carica l'indice in BX
ADD BX,BX    ' moltiplica per due, in quanto ogni
             ' elemento è lungo due byte
MOV AX,[BX+offset] ' tieni conto dell'offset del primo
             ' elemento dell'array
```

È evidente che se l'indice caricato in BX è maggiore del valore di UBOUND(a), in AX sarà caricato il contenuto di una cella di memoria che non appartiene al vettore. La situazione è ancora più pericolosa quando si *assegna* un valore ad un elemento inesistente: in tal caso, infatti, il programma modifica il valore di un'altra variabile o di un descrittore stringa, e può mandare in tilt il programma con un errore di tipo "String Space Corrupt". In questo caso si tratta di un bug molto pericoloso e difficile da scoprire, in quanto i suoi effetti non si manifestano immediatamente, ma solo quando il programma tenta di utilizzare il valore nella locazione di memoria alterata erroneamente, oppure al successivo garbage collection delle stringhe.

In definitiva, è miglior consiglio per evitare queste situazioni è di eseguire il test completo del programma *prima* di compilare, dall'interno dell'ambiente integrato; se però per qualche motivo il programma non può essere testato in modo interprete, si deve ricorrere alla compilazione con l'opzione /D, che ottiene i seguenti effetti:

1. tutte le moltiplicazioni e divisioni su interi avvengono mediante chiamata a subroutine, per cui viene indirettamente attivato il controllo degli overflow
2. gli accessi agli elementi degli array avvengono anch'essi attraverso chiamate a subroutine, che controllano che l'indice sia compreso nell'intervallo valido

3. dopo ogni istruzione e all'interno delle INPUT il BASIC controlla se l'operatore ha premuto il tasto CTRL-BREAK; in tal modo è possibile bloccare l'esecuzione di un programma che entra in un loop infinito

È evidente che l'opzione /D rallenta notevolmente l'esecuzione dei programmi compilati, per cui essa dovrebbe essere evitata nella versione finale di distribuzione degli applicativi.

Se il programma lavora con array huge maggiori di 64K, e quindi deve essere compilato con l'opzione /AH, non ci si deve preoccupare del controllo al punto 2, in quanto l'opzione /AH fa sì che l'accesso agli elementi degli array sia regolamentato da una subroutine, che provvede automaticamente a controllare la validità dell'indice.

LA FUNZIONE INKEY$ E LE TASTIERE ESTESE

Questo è il bug che più di tutti mi ha fatto penare, ed è causato da un problema di compatibilità del BASIC con alcuni dei ROM BIOS montati sui primi IBM AT e sui primi cloni di quest'ultimo; accade infatti che su alcune di queste macchine la funzione INKEY$ restituisce sempre una stringa nulla. Il servizio di assistenza tecnica Microsoft U.S.A., da me interpellato al proposito, ha riconosciuto il problema, affermando che il BASIC "...*is* BIOS-*dependant*", ma non ha fornito alcuna soluzione, oltre a quella - ovvia - di sostituire il ROM BIOS.

Il BIOS fornisce due set completamente distinti di servizi per la lettura dei tasti premuti; un insieme di funzioni per leggere i tasti di una tastiera normale ed un set parallelo per la tastiera estesa. E' ovvio che un programma dovrebbe usare il set appropriato alla tastiera collegata al computer su cui viene eseguito: utilizzando il primo set su una tastiera estesa non sarebbe infatti possibile leggere i tasti F11-F12, mentre utilizzando il secondo set con una tastiera normale non riceverebbe alcun codice di scansione. Questo secondo comportamento è proprio quello manifestato dai programmi BASIC, per cui il problema sembra dovuto al fatto che in alcuni casi le routine di startup identificano erroneamente il tipo di tastiera, e indicano si usare il set di istruzioni per una tastiera estesa che però non è supportata dal BIOS.

Per una soluzione definitiva al problema occorre evitare del tutto le istruzioni di I/O del BASIC (cioè INKEY$ e LINE INPUT), e costruire delle routine equivalenti. Per fortuna non è necessario ricorrere ad una routine in linguaggio macchina, poiché per accedere alle routine del BIOS è sufficiente l'istruzione INTERRUPT. Ecco la routine che ho preparato:

```
FUNCTION INKEY2$ ()
'___________________
'  Leggi un tasto
'___________________
 DIM reg AS RegType
reg.ax = &H100   ' servizio 1 in AH - controlla che un
Interrupt &H16, reg, reg ' tasto sia presente nel buffer
 IF reg.flags AND 64 THEN ' se restituisce il flag zero attivo
  INKEY2$ = ""   ' non è stato premuto alcun tasto
 ELSE
  reg.ax = 0   ' altrimenti leggi il tasto
  Interrupt &H16, reg, reg
  IF reg.ax AND 255 THEN
   ' se AL<>0 si tratta di un tasto del set normale
   INKEY2$ = CHR$(reg.ax AND 255)
  ELSE
   ' altrimenti si tratta di un tasto del set esteso e
   ' AH contiene il suo codice di scansione
   INKEY2$ = MKI$(reg.ax)
  END IF
 END IF
END FUNCTION
```

Usando la routine INKEY2$ per costruire una routine di input controllato è possibile evitare l'uso delle istruzioni INPUT e LINE INPUT.

I BUG DEL VISUAL BASIC PER DOS

Il nucleo del Visual BASIC per Dos non è molto dissimile dal BASIC PDS, e difatti molti dei bug presenti in quest'ultimo sono stati "ereditati" dall'ultima versione di questo linguaggio; d'altra parte, la grande innovazione del VBDOS riguarda il Form Generator e l'ambiente di programmazione event-driven; inevitabilmente anch'essi soffrono di alcuni problemi, per fortuna facilmente risolvibili:

- il manuale avverte che per produrre messaggi al di fuori dei form è necessario eseguire una istruzione SCREEN.HIDE; non è però indicato che, all'interno dei moduli form (.FRM), i comandi PRINT e CLS sono considerati metodi che agiscono sul form, ed essendo il form stesso invisibile i comandi di questo tipo apparentemente non hanno effetto alcuno. La soluzione è di eseguire tutte le istruzioni di questo tipo da un modulo normale (.BAS); se l'azione deve avvenire dall'interno di una event procedure è necessario costruire una subroutine in un file .BAS e richiamare la subroutine dalla event procedure
- esiste una limitazione alle azioni che possono essere intraprese nella procedura **Click** di un menù non terminale, ossia di un menù a cui è collegato un altro menù di livello inferiore: in generale, è consigliabile limitarsi a modificare le proprietà del sottomenù (per esempio, per abilitare o disabilitare alcune voci), e si dovrebbe evitare l'attivazione

indiretta di altri eventi con DOEVENTS o SETFOCUS, o modificare le proprietà dei menù di livello superiore; in caso contrario i risultati non sono prevedibili

- esiste un limite al numero di volte in cui un controllo appartenente a un custom control array può essere caricato e scaricato dinamicamente a runtime; questo avviene perché parte della memoria usata dal controllo non è rilasciata dal Visual BASIC fino a quando non termina la procedure corrente (normale o *event procedure*), oppure la dialog box modale corrente. A lungo andare la memoria libera diminuisce sotto il limite consentito e il programma termina con un errore "Out of Memory"
- se un form contiene un controllo alto quanto l'altezza del form stesso - quindi il controllo è posizionato alle coordinate (0,0) - e in seguito si aggiunge una barra di menù (diminuendo quindi l'altezza effettiva del form) il controllo diventa invisibile e quindi inattivo; lo stesso accade quando il form contiene già una barra dei menù, ma il form è ridimensionato in modo tale che la barra dei menù deve occupare più di un singolo rigo di schermo

Differenze con Visual BASIC per Windows:

- se un controllo è disabilitato, un click del mouse non attiva la procedura **Click** del form o della picture box che contiene il controllo (come dovrebbe avvenire e come infatti accade in Visual BASIC per Windows)
- nelle dialog box prodotte dal comando o dalla funzione **MSGBOX**, è necessario premere il tasto ALT per attivare uno dei bottoni di comando; in Visual BASIC per Windows, e in tutti i programmi Windows, è sufficiente premere il tasto evidenziato, senza il tasto ALT
- in VBDOS l'evento **KeyUp** avviene subito dopo l'evento **KeyDown**, anche se l'operatore non ha ancora rilasciato il tasto; per questo motivo non è possibile, ad esempio, monitorare questi due eventi per calcolare per quanto tempo il tasto è stato premuto; in Visual BASIC per Windows l'evento **KeyUp** è generato soltanto quando il tasto è realmente rilasciato
- in Visual BASIC per Dos tutte le listbox, i combo box e gli altri controlli di questo tipo possiedono una scroll bar verticale, indipendentemente dal numero di elementi contenuti; questo comportamento differisce dalla versione per Windows, in cui la scroll bar appare solo se il numero di elementi supera quello visualizzabile in un determinato momento
- il contenuto delle file box e delle directory box non è ordinato, ma rispecchia l'ordine con cui i nomi di file sono memorizzati nella directory corrente su disco; questo comportamento differisce dal Visual BASIC

per Windows, in cui il contenuto di questi controlli è sempre ordinato automaticamente in ordine alfabetico

PROBLEMI IN QBX E VBDOS

Gli ambienti di sviluppo QBX e VBDOS presentano dei problemi di compatibilità con alcune tastiere non americane tra cui, è quasi superfluo dirlo, anche la tastiera italiana. Il problema è evidente con i tasti che sono prodotti ottenuti tenendo premuto il tasto **AltGr:**

@ [ ]

è evidente che il problema è abbastanza fastidioso, soprattutto in quanto coinvolge i caratteri "#" e "@" che in BASIC sono utili per definire le variabili DOUBLE e CURRENCY. Il problema è stato riconosciuto da Microsoft, che però non fornisce alcuna soluzione e si limita a suggerire di comporre il codice del carattere desiderato sul tastierino numerico, tenendo premuto il tasto ALT. I codici ASCII dei caratteri in questione sono:

#	35	[	91
@	64	]	93

Come se non bastasse, in Visual BASIC per Dos questa tecnica funziona correttamente solo se il tasto NumLock è attivato, mentre in tutte le altre versioni del BASIC la combinazione ALT+tasto funziona sempre indipendentemente dallo stato di NumLock.

Un ulteriore bug, che non appare nelle *buglist* ufficiali, influenza la combinazione CTRL+\. che stando alla documentazione e alle shortcut key associate al menù **Search**, dovrebbe attivare la ricerca del testo evidenziato, ma che invece causa l'inserimento del carattere "\", esattamente come se il tasto CTRL non fosse affatto premuto; questo bug è tanto più fastidioso in quanto il carattere "\" va a sostituire il testo evidenziato, che sarebbe perso definitivamente se non fosse per la possibilità di impartire un comando di **Undo**.

IL COMPILATORE E IL LINKER

Esiste una differenza importante tra i programmi interpretati e i programmi compilati: nei primi a ciascuna SUB o FUNCTION è allocato uno spazio di memoria di 64K, mentre nei secondi ciascun modulo deve essere compreso

in 64K. A causa di questa differenza alcuni programmi che funzionano perfettamente sotto QB/QBX/VBDOS provocano il seguente errore al momento della compilazione:

```
Program Memory Overflow
```

La soluzione, ovvia, è di ripartire le procedure in due o più moduli distinti.

In alcuni casi il compilatore fornisce degli inpiegabili messaggi "Internal Error" che di fatto impediscono la compilazione di un programma altrimenti corretto; se sperimentate questo tipo di problema - presumibilmente causato dal tentativo di ottimizzare la sequenza di alcune operazioni - provate ad aggiungere una etichetta oppure un numero di linea alla riga che provoca l'errore. Per sapere qual è la riga in questione, chiedete al compilatore di produrre un *list file* e controllate in seguito l'ultima riga presente nel list file prima dell'errore.

Il linker in dotazione al VBDOS tenta di sfruttare la memoria espansa o estesa per migliorare le prestazioni; quando però la quantità di memoria libera EMS o XMS è scarsa - al di sotto dei 200K - il linker si blocca con il seguente messaggio di errore:

```
R6907 DOSX16 Not enough memory on exec
```

Per evitare questo errore occorre usare lo switch /R che, per essere riconosciuto correttamente, deve essere il primo switch sulla riga di comando per il linker; il problema si manifesta solo nelle compilazioni dal prompt del Dos o da file batch, poiché l'ambiente di sviluppo VBDOS riconosce la eventuale scarsità di memoria espansa o estesa, per cui questo switch è aggiunto automaticamente, se necessario, quando si compila un programma con il comando Make EXE File.

Quando si compila e si linka dall'interno del VBDOS una eventuale opzione /SEG è ignorata; questo avviene perché il Visual BASIC calcola in ogni caso il valore più idoneo, verificando il numero dei moduli, sommando i segmenti richiesti dall'eventuale gestore ISAM e aggiungendo circa 50 segmenti per sicurezza; normalmente il valore risultante è sufficiente, ma in alcuni casi - in particolare quando si linka una libreria composta da molte routine - esso si rivela non abbastanza grande, e il processo di link termina con un messaggio:

```
L1049  Too many segments
```

l'unico modo di aggirare questa limitazione del BASIC è di compilare il programma dal prompt del Dos.

Un ultimo bug del linker Microsoft: quando la memoria RAM è insufficiente oppure la tavola dei simboli è troppo grande e si rende necessario creare un file temporaneo, il linker NON controlla che lo spazio su disco sia sufficiente, e procede senza messaggi di errore fino al completamento del processo e alla creazione del file eseguibile, che però è corrotto e solitamente manda

immediatamente in crash il sistema; il bug è tanto più subdolo in quanto al termine del linking il file temporaneo (che può essere grande anche alcune centinaia di Kbyte) viene normalmente cancellato, per cui l'operatore non è immediatamente insospettito da una scritta del tipo "0 bytes free" dando un comando DIR o CHKDSK. Non ho trovato traccia di questo bug nella documentazione Microsoft, né il servizio di assistenza italiana sembrava esserne a conoscenza.

STRANEZZE VARIE

Termino questo capitolo con una raccolta "assortita" di problemi non facilmente catalogabili in alcun modo.

- attenzione alla priorità degli operatori aritmetici, che non sempre corrisponde a quella a cui siamo abituati; ad esempio la seguente espressione:

```
PRINT -4 ^ 2
```

  stampa il valore -16, e non 16 come ci si potrebbe aspettare; per ottenere il quadrato di un numero negativo siamo costretti ad usare le parentesi:

```
PRINT (-4) ^ 2
```

- il valore restituito dalla funzione COMMAND$ resta immutato anche quando il controllo passa ad un secondo programma BASIC richiamato con il comando RUN; in altre parole, se il programma PROVA.EXE è richiamato dal prompt del Dos con il comando:

```
prova test
```

  e PROVA.EXE richiama a sua volta il programma PROVA2.EXE con il comando:

```
RUN "prova2 pippo"
```

  la funzione COMMAND$ restituirà la stringa "TEST" sia quando è richiamata all'interno di PROVA.EXE che di PROVA2.EXE

- in caso di errore su una stampante parallela e con un programma eseguito sotto MsDos 5 o superiore, la funzione ERDEV$ restituisce una stringa errata; infatti, invece di restituire la stringa "LPT1:" restituisce la stringa "M:", e invece di "LPT2:" restituisce "T:"; questo bug riguarda tutte le versioni del BASIC; l'unico modo di aggirare il problema è di tenere traccia in qualche altro modo della periferica che sta effettuando l'output, conservandone il nome in una variabile globale accessibile dalla routine che tratta gli errori

- il BASIC segnala un errore "Expression Too Complex" quando si tentano di concatenare più di 19 stringhe; la cosa strana è che in alcuni casi il

QuickBASIC e il compilatore BC 7 segnalano questo errore indicando la riga precedente a quella in questione, e ciò complica il lavoro di debug. Inoltre, e questa è davvero una piccola perla, in alcuni casi si continua a ottenere questo errore nell'ambiente di sviluppo anche se la riga in questione è stata "commentata" con una REM. Nei programmi compilati dal prompt del Dos con il compilatore fornito con BASIC PDS e Visual BASIC per Dos, un errore di sintassi in una istruzione UBOUND - ad esempio, UBOUND(array()) - provoca un errore:

```
Run-time error R6000 - stack overflow
```

e il blocco dell'intero sistema.

- come avverte la documentazione, non è consigliabile usare nella stessa istruzione un comando e una funzione che eseguono input e output da file, anche da due file differenti; ad esempio, la seguente istruzione dovrebbe essere evitata:

```
PRINT #1, INPUT$(10, #2)
```

come pure dovrebbero essere evitati i riferimenti a FUNCTION scritte in BASIC che al loro interno eseguono input da file, come in:

```
PRINT #2, LeggiDaFile$(30)
```

non rispettando questa regola i risultati sono imprevedibili (ad es. in alcuni casi l'ouput della PRINT è inviato allo schermo anziché su file). La soluzione, abbastanza semplice, è quella di assegnare il risultato della funzione che esegue l'output ad una variabile temporanea:

```
temp$ = INPUT$(10, #2)
PRINT #2, temp$
```

- la funzione FRE(-3) restituisce il numero di kilobyte liberi in memoria EMS; ma non è immediatamente chiaro che questa forma della funzione FRE è un semplice aiuto nella scrittura dei programmi interpretati e non può essere usata nei programmi compilati, in quanto produce un errore "Feature Unavalaible"; per testare la quantità di memoria espansa libera nei programmi compilati è necessario eseguire una chiamata diretta al driver EMS, come spiegato altrove in questo libro

- può essere utile sapere che durante l'esecuzione delle procedure per il trattamento degli errori (attivate da ON ERROR), sono sospese tutte le altre procedure per il trattamento degli eventi (attivate da ON KEY, ON TIMER, ecc.); gli eventi sono riabilitati subito dopo l'esecuzione di una istruzione RESUME o RESUME NEXT

- nei programmi compilati con l'opzione /FPA (il package alternativo per l'aritmetica floating point), applicando la funzione VAL ad una stringa del tipo "E999" (una serie di nove preceduta da una "E") si provoca un errore di overflow; se invece si compila senza /FPA si ottiene correttamente un valore nullo. Attenzione: se compilate un modulo con l'opzio-

ne /FPA ricordarsi di usare la stessa opzione per tutti i moduli della applicazione; in caso contrario il programma visualizzerà un messaggio:

```
Error during run-time initialization
```

e il sistema sarà bloccato. Per lo stesso motivo, non è permesso creare Quick Library utilizzando il package alternativo.

- se si tenta di cancellare un file su un computer remoto collegato in rete locale, e il programma non ha il diritto di accesso al file da cancellare, il BASIC emette un errore "File Not Found" anziché "Access Denied" come sarebbe più logico. Per cui, in casi come questi, per sapere cosa è realmente accaduto occorre testare l'esistenza del file in qualche altro modo, ad esempio con la funzione DIR$ (che però non è disponibile in QuickBASIC)
- quando si inviano dei dati alla stampante, il BASIC emette automaticamente una coppia di caratteri Carriage Return/Line Feed (ASCII 13 + ASCII 10) quando la larghezza della riga supera la dimensione di default, che è di 80 colonne. Si tratta di un retaggio dei vecchi tempi in cui le stampanti non erano abbastanza intelligenti da mandare a capo da sole la testina di stampa quando raggiungeva il margine destro. Per poter stampare a più colonne (ad es. utilizzando il compresso e/o una stampante a carrello largo), occorre pertanto dare un comando WIDTH LPRINT:

```
WIDTH LPRINT 132
WIDTH LPRINT 255  ' disattiva del tutto il controllo
```

inoltre il BASIC invia automaticamente un carattere Line Feed dopo ogni Carriage Return fornito dal programma, la qual cosa è molto fastidiosa quando si lavora con la stampante in modo grafico; in questo caso, per disabilitare questa funzione occorre aprire un file di output con:

```
OPEN "LPT1:BIN" FOR OPEN AS #1
```

ed inviare i dati con una istruzione:

```
LPRINT #1, bytes$; ' notare il punto e virgola finale
```

- l'istruzione PAINT ha dei problemi nell'ambiente QBX e col compilatore BC7 quando si specifica come motivo per lo sfondo la stringa CHR$(0); ecco un esempio di codice che provoca il problema:

```
SCREEN 8
LINE (0,0)-(150,150), 15,B
LINE -(0,0),14
PAINT (1,1), 15, chr$(0)
```

- sempre a proposito della grafica, nelle versioni 4.0 e 4.5 del QuickBASIC l'opzione BF dell'istruzione LINE non rispetta la finestra di clipping creata con il comando VIEW; ad esempio:

```
SCREEN9
VIEW (1,1)-(150,150)   ' imposta una finestra di clip
LINE (-50,50)-(-100,100),12,B  ' il cliping è corretto
LINE (-50,50)-(-100,100),12,BF ' il clip NON è corretto
```

- su alcuni computer, in particolare quelli dotati di microprocessore 80286 o superiore, le funzioni che il BASIC mette a disposizione per controllare lo stato del joystick possono non funzionare perfettamente; questo accade perché su questi computer il BASIC accede al joystick attraverso una chiamata al BIOS, mentre con i processori della generazione precedente (8088/8086/80186) il BASIC si interfaccia direttamente all'hardware. Capita però che alcuni BIOS non gestiscano perfettamente i segnali sulle porte di I/O, e quindi che su queste macchine queste funzioni BASIC non funzionino correttamente.

- infine, ecco una lista di schede video che non funzionano, o per lo meno possono creare problemi con i programmi BASIC e con gli ambienti QB/QBX/VBDOS:

  Techmar VGA

  Quadram VGA

  Vega-Video 7 Fastwrite VGA

  VegaVGA

  Compaq Laptop

  Genoa SuperVGA Hires 5100 e 5200

  VIP ATi VGA

  Sigma EGA

11

LA RICORSIONE

La ricorsione è una tecnica potentissima che spesso viene trattata molto distrattamente dalla maggior parte dei manuali di programmazione. Come è mia abitudine, non mi dilungherò troppo nella teoria e affronterò subito il lato pratico della questione.

Una funzione si dice *ricorsiva* quando essa compare nella sua stessa definizione. La funzione ricorsiva più conosciuta, nonché quella presa sempre ad esempio quando si parla di programmazione ricorsiva, è la funzione fattoriale, che infatti può essere definita in questo modo:

$$fattoriale(0) = 1$$

$$fattoriale(n) = n * fattoriale(n-1) \qquad per\ n >= 1$$

Parallelamente alle funzioni ricorsive, una procedura si dice *ricorsiva* quando richiama se stessa. Sempre facendo riferimento al calcolo del fattoriale, è facile vedere come passare dalla definizione di una funzione ricorsiva alla sua traduzione in un qualsiasi altro linguaggio di programmazione che accetta la ricorsione: ecco una procedura BASIC che calcola il fattoriale

```
FUNCTION Factorial# (number%)
        '________________________
        '  Fattoriale di un numero - versione ricorsiva
        '________________________
        IF number% = 0 THEN
                Factorial# = 1
        ELSE
                Factorial# = number% * Factorial#(number% - 1)
        END IF
END FUNCTION
```

In realtà, la funzione fattoriale, nonostante la sua utilità e la sua onnipresenza nei vari testi sulla programmazione, non è un esempio particolarmente indicativo della utilità di questa tecnica, in quanto è facile scrivere un programma Basic equivalente che porta allo stesso risultato senza usare la ricorsione, e soprattutto molto più velocemente.

```
FUNCTION Factorial2# (number%)
        '____________________________
        ' Fattoriale di un numero - versione non ricorsiva
        '____________________________
        DIM result#, index%
        result# = 1
        FOR index% = 1 TO number%
                result# = result# * index%
        NEXT
        Factorial2# = result#
END FUNCTION
```

Questo significa che la ricorsione non è strettamente necessaria ?

La risposta a questa domanda è a metà strada fra un secco "sì" e un ponderato "no": in effetti la ricorsione *non* è indispensabile, ed è stato provato che qualunque programma ricorsivo può essere trasformato in un programma equivalente che però non sfrutta la ricorsione; è anche vero che in generale questa trasformazione non è affatto banale e che *in pratica* per problemi (e algoritmi) ricorsivi di media o grande complessità, la ricorsione rimane l'unico metodo per ottenere risultati accettabili. E' anche vero, d'altra parte, che le tecniche ricorsive sono sicuramente da evitare quando vogliamo ottenere le migliori prestazioni da un programma; ad esempio, tra le due implementazioni dello stesso algoritmo che abbiamo appena visto, la funzione non ricorsiva è in media il 60% più veloce di quella ricorsiva.

In definitiva è conveniente cercare di evitare la ricorsione quando possibile, e di usarla con oculatezza quando il tipo del problema è intrinsecamente ricorsivo ed appare sconsigliabile cercare un algoritmo non ricorsivo equivalente. La teoria ci dice anche che in presenza della cosiddetta *tail recursion* (da tradurre con "ricorsione di coda", anche se non ho mai trovato questo termine su un testo italiano) la traduzione in un programma non ricorsivo è facile ed immediata. La *tail recursion* si verifica quando nella definizione dell'algoritmo la chiamata ricorsiva appare appunto in coda; una occhiata alla definizione del fattoriale (e al programma da esso derivato) ci fa vedere come il fattoriale appartenga a questa categoria di funzioni e ciò spiega la facilità con cui abbiamo potuto ottenere una versione non ricorsiva dello stesso.

Un'altro esempio di tail recursion è dato dal calcolo del numero di FIbonacci; per chi non lo sapesse o non se lo ricordasse, il numero di Fibonacci di un dato ordine N è così definito:

$$Fibonacci(0) = 1$$

$$Fibonacci(1) = 1$$

$$Fibonacci(n) = Fibonacci(n\text{-}2) + Fibonacci(n\text{-}1)$$

il seguente listato riporta la versione ricorsiva, che è però di una lentezza estenuante e in pratica può essere usata solo per numeri molto piccoli: lascio ai più volenterosi il compito di scrivere una versione equivalente che non sfrutti la ricorsione.

```
FUNCTION Fibonacci# (ordine%)
     '____________________________
     '   Numero di Fibonacci - versione ricorsiva
     '____________________________
     IF ordine% = 0 OR ordine% = 1 THEN
            Fibonacci# = 1
     ELSE
            Fibonacci# = Fibonacci#(ordine%-1) + Fibonacci#(ordine%-2)
     END IF
END FUNCTION
```

LO STACK DEL PROGRAMMA

Oltre alla minore velocità, le procedure ricorsive soffrono di un altro difetto, ossia di richiedere spesso molto spazio sullo stack, che viene usato non solo per conservare gli indirizzi di ritorno ma anche tutte le variabili locali. Di regola ogni chiamata ad una procedura richiede quattro byte sullo stack più due byte per ogni argomento, ma se ad esempio la procedura dichiara come variabili locali un intero, un numero in doppia precisione e una STRING * 80 caratteri, ogni livello di ricorsione richiede più di un centinaio byte: è facile vedere che dopo una trentina di ricorsioni lo spazio sullo stack di default (3K) si esaurisce: se il programma è eseguito in modo interpretato nell'ambiente di sviluppo l'errore viene intrappolato dal Basic, ma nel caso dei programmi compilati esso provoca un crash di sistema, tanto grave che nella maggior parte dei casi non funziona neanche la combinazione dei tasti Ctrl-Alt-Canc !

Per fortuna la soluzione è abbastanza semplice, in quanto l'istruzione CLEAR (e l'istruzione STACK in QBX e VBDOS) permette di modificare la dimensione di default dello stack, ad es.

```
CLEAR , , 6000          ' QB, QBX e VBDOS
```

oppure

```
STACK  6000             ' solo con QBX e VBDOS
```

ricordatevi che la dimensione minima dello stack è di 350 byte circa, mentre la dimensione massima dipende dal programma in questione, e può essere calcolata (ma solo in QBX e VBDOS, non in QuickBASIC) grazie alla funzione

STACK. Per allocare la maggior quantità possibile di memoria per lo stack si deve eseguire l'istruzione:

```
STACK  STACK            ' solo con QBX e VBDOS
```

E' importante sapere che lo spazio allocato per lo stack viene sottratto a DGROUP, e quindi alle variabili semplici, alle stringhe near ed ai vettori STATIC. Per questo motivo è importante non allocare più memoria del necessario. A tale scopo, in fase di test si può utilizzare la funzione FRE(-2), che restituisce appunto il numero di byte usati sullo stack dal programma; ecco come procedere:

```
CLEAR , , 10000
 esegui il programma
stackUsed = 10000 - FRE(-2)
```

Al termine del programma la variabile **stackUsed** conterrà proprio la quantità di byte di stack effettivamente utilizzati dal programma, a cui conviene aggiungere 1K per una maggiore sicurezza. La funzione FRE(-2) è inoltre utile nei programmi compilati, per tenere sotto controllo lo stato dello stack. Per esempio, all'inizio di una funzione ricorsiva potremmo inserire il seguente controllo:

```
IF FRE(-2) < 500 THEN ERROR 200
```

Non eviteremo un errore, ma almeno si tratterà di un errore "intrappolabile" dal BASIC, in modo che il nostro programma potrà agire di conseguenza. Un'altra possibilità è di compilare il programma con l'opzione /D, che tra le varie funzioni ha anche quella di controllare gli overflow dello stack. Otterremo un programma più lento e meno compatto ma l'alternativa, come ho già detto, è un blocco totale del sistema, sicuramente più sgradito e da evitare a tutti i costi.

VARIABILI STATICHE E DINAMICHE

Come ben sappiamo, una funzione può essere definita STATIC oppure DYNAMIC; di regola tutte le procedure e funzioni sono dinamiche, a meno che l'attributo STATIC non sia esplicitamente dichiarato nella definizione della funzione. Come suggeriscono i manuali del linguaggio, le dichiarazioni STATIC accelerano leggermente la velocità di esecuzione, in quanto in questo caso le variabili locali sono conservate in DGROUP anziché sullo stack: tradotto in Assembly, questo significa che le variabili STATIC sono lette o scritte mediante indirizzamento assoluto, mentre le variabili dinamiche richiedono l'indirizzamento indicizzato (rispetto al registro BP) che impiega qualche ciclo di clock in più. Non solo, ma per quanto detto in precedenza, le variabili dinamiche occupano spazio sullo stack, e aumentano la probabilità di uno stack overflow.

Nonostante questo avvertimento, è opportuno chiarire subito che tutte le procedure e funzioni ricorsive devono essere dinamiche e in generale non devono contenere dichiarazioni di variabili locali STATIC. Il motivo dovrebbe essere evidente: se una variabile fa riferimento sempre alla stessa locazione in DGROUP, il sua valore verrà alterato da una chiamata ricorsiva alla stessa procedura. Ovviamente ci sono delle eccezioni, e cercheremo di illustrarle a dovere.

La prima eccezione si ha quando occorre conservare in una variabile il livello di ricorsione; è evidente che tale informazione non può essere conservata in una variabile dinamica, che è automaticamente azzerata ad ogni invocazione della procedura. Il programma del listato 4 sfrutta la ricorsione per disegnare l'albero delle directory, a partire dalla directory radice; come sappiamo, il Basic 7.x e VBDOS (ma non il QuickBASIC) dispone dell'istruzione DIR$ per ottenere l'elenco dei file in una directory, ma non delle directory, per cui dovremo ricorrere al trucco di eseguire una istruzione DIR e filtrare con FIND le righe che si riferiscono alle directory. In questo caso il livello di ricorsione è necessario per indentare correttamente i nomi delle directory: ogni livello di directory è spostato di tre spazi a destra rispetto al livello immediatamente superiore; in realtà, il problema può essere risolto più elegantemente richiamando direttamente il Dos, ma in questo caso mi preme esclusivamente illustrare la tecnica ricorsiva.

```
DEFINT A-Z
SUB ShowDirTree
	'______________________________
	'  Mostra l'albero delle directory, indentando
	'  le directory di livello inferiore
	'______________________________
	STATIC recursionLevel
	$STATIC handle, i$
	DIM dirlist$(50), dirnum, i
	recursionLevel = recursionLevel + 1

	' crea un file temporaneo con l'elenco delle directory
	SHELL "DIR|FIND " + CHR$(34) + "<DIR>" + CHR$(34) + ">dirs.$$$"
	handle = FREEFILE
	OPEN "dirs.$$$" FOR INPUT AS #handle

	' leggi i nomi delle directory nell'array dirlist$()
	dirnum = 0
	DO UNTIL EOF(handle)
		LINE INPUT #handle, i$
		' non tenere conto delle directory "." e ".."
		IF ASC(i$) <> 46 THEN
			i$ = RTRIM$(LEFT$(i$, 12))
			IF LEN(i$) > 8 THEN
				' la directory possiede una estensione
				i$ = RTRIM$(LEFT$(i$, 8)) +"." +RTRIM$(RIGHT$(i$,3))
			END IF
			dirnum = dirnum + 1
			dirlist$(dirnum) = i$
		END IF
```

```
        LOOP
        CLOSE #handle
        KILL "dirs.$$$"

        ' per ciascuna sottodirectory, mostra il nome e richiama
        ' ricorsivamente la procedura - il livello di ricorsione
        ' determina l'indentazione e crea l'effetto "albero"
        FOR i = 1 TO dirnum
                PRINT SPACE$(recursionLevel * 3); dirlist$(i)
                CHDIR dirlist$(i)
                ShowDirTree
                CHDIR ".."
        NEXT
        recursionLevel% = recursionLevel% - 1
END SUB
```

Il programma precedente mostra un altra eccezione alla regola sull'uso delle variabili dinamiche: se il valore della variabile non deve essere conservato durante le chiamate ricorsive, la variabile stessa può essere dichiarata come STATIC: nella procedura ShowDirTree questo è il caso delle variabili handle e i$,che infatti sono dichiarate come statiche, non occupano inutilmente spazio sullo stack durante le chiamate ricorsive ed in generale accelerano l'esecuzione.

UN ESEMPIO "CLASSICO": LE TORRI DI HANOI

Molti di voi conosceranno il rompicapo delle Torri di Hanoi, anche se è sicuramente più famoso per quanto se ne è scritto sui libri di programmazione che per la sua reale diffusione. In poche parole, nella sua forma originaria, il rompicapo consiste in una piattaforma con tre pioli e una serie di N dischi di dimensioni differenti e forati al centro, che possono essere quindi "inpilati" sui pioli. Il gioco consiste nel costruire una piramide di dischetti sul primo piolo (chiamiamolo "A") e di trasportare la piramide su un'altro dei due pioli rimanenti (chiamiamolo "B") stando attenti a rispettare le seguenti regole:

a) si può spostare solo un disco per volta, da un piolo all'altro; non possono esistere dischetti "liberi" non impilati su un piolo, se non per il tempo necessario allo spostamento

b) non è possibile che un dischetto poggi su un altro dischetto di dimensioni inferiori

Quando il numero N dei dischetti è pari ad uno, il problema è banale: basta spostare il dischetto da A a B ! Con due dischetti si è costretti ad usare il terzo piolo (chiamiamolo "C") per "parcheggiare" temporaneamente il dischetto più piccolo. La sequenza corretta sarà allora la seguente

sposta il dischetto piccolo da A a C

sposta il dischetto grande da A a B

sposta il dischetto piccolo da C a B

Cosa accade col crescere del numero dei dischetti ? Con N = 3 il problema è ancora abbastanza facilmente risolvibile, e si risolve con la seguente sequanza di sette operazioni

sposta il dischetto piccolo da A a B

sposta il dischetto medio da A a C

sposta il dischetto piccolo da B a C

sposta il dischetto grande da A a B

sposta il dischetto piccolo da C a A

sposta il dischetto medio da C a B

sposta il dischetto piccolo da B a A

quando però il numero dei dischetti aumenta ulteriormente le cose diventano troppo complicate, ed è necessario ricorrere al computer. Ed è a questo punto che entra in ballo la ricorsione: è facile vedere che il problema è di tipo ricorsivo, anche se di tipo differente da quelli visti finora. Infatti, il problema di spostare N dischi da A a B si riduce a tre sottoproblemi

1. spostare gli N-1 più piccoli dischi da A a C (il piolo temporaneo)
2. spostare il disco rimanente (il più largo) da A a C
3. spostare gli N-1 più piccoli dischi da C a B

dove il problema (2) è banale e (1) e (3) sono simili al problema originale ma più semplici. Ecco una procedura BASIC ricorsiva che risolve elegantemente il problema

```
SUB Hanoi (dischi%, sorgente$, destinazione$, intermedio$)
        IF dischi% = 1 THEN
                PRINT "Sposta un disco da "; sorgente$; " a "; destinazione$
        ELSE
                Hanoi dischi% - 1, sorgente$, intermedio$, destinazione$
                Hanoi 1, sorgente$, destinazione$, ""
                Hanoi dischi% - 1, intermedio$, destinazione$, sorgente$
        END IF
END SUB
```

Per testare la procedura, provate a richiedere la lista di operazioni necessarie per spostare tre dischi dal piolo A al piolo B usando C come piolo intermedio

```
CALL Hanoi (3, "A", "B", "C")
```

UN ESEMPIO COMPLESSO: LE DERIVATE SIMBOLICHE

Per chiudere in bellezza con qualcosa che renda realmente giustizia a questa potente tecnica di programmazione, ecco a voi un bel programmone per il calcolo di derivate simboliche. Cosa ha a che fare il calcolo della derivata di una funzione con la ricorsione ? Prima di tutto la derivata di ordine N-esimo di una funzione può essere scritta in modo ricorsivo:

derivata n-esima di f(x) = derivata [derivata (n-1)-esima di f(x)]

ma qual che più ci interessa è che molte regole di derivazione sono esse stesse implicitamente ricorsive; ad esempio, la derivata della somma di due funzioni è definita come

derivata [f(x) + g(x)] = derivata[f(x)] + derivata[g(x)]

mentre la derivata del prodotto di due funzioni si calcola come

derivata [f(x) * g(x)] = derivata[f(x)] * g(x) + f(x) * derivata[g(x)]

Ce n'è abbastanza per qualificare la derivazione come una operazione di tipo ricorsivo, ed infatti il programma che segue sfrutta pesantemente la ricorsione per semplificare la struttura del codice; sebbene si tratti di oltre 300 righe di codice, va tenuto presente che il calcolo della derivata simbolica non è affatto un compito banale. Uno degli aspetti non secondari di un programma che manipola espressioni algebriche e di semplificare, o per lo meno di tentare di semplificare, le operazioni algebriche; questo è quanto fa nel suo piccolo anche il programma presentato in queste pagine: si veda ad esempio le routine add$, sub$, mul$ e div$, che trattano correttamente i casi banali (ad esempio la moltiplicazione per l'unità oppure la somma con un addendo nullo). Un semplificatore algebrico appena più sofisticato di questo richiederebbe diverse migliaia di righe di codice, in quanto dovrebbe memorizzare numerose regole (ad es. le identità come log(e^x)=x oppure sin(x)^2+cos(x)^2 = 1) e un parser potentissimo per il pattern matching; non è un caso che programmi come Mathematica siano prodotti apparsi solo recentemente su personal computer, anche se i loro predecessori (come Macsima) operano da più di quindici anni sulle workstation o addirittura sui mainframe.

Vi lascio allo studio del listato del programma, reso più arduo del solito dal fatto che, per motivi di spazio, non ho potuto abbondare con i commenti; se tentate di eseguire il trace di un programma fortemente ricorsivo come questo imparerete presto quanto è facile perdersi nei meandri della ricorsione. I più pigri potranno trascriverlo così com'è e, perché no, usarlo magari come "aiuto" in ambito scolastico.

```
'________________________________________
'                 CALCOLO  DERIVATE  SIMBOLICHE
'
'                   1989-94  Francesco Balena
'________________________________________

DEFINT A-Z
DECLARE SUB EliminaSpazi (a$)
DECLARE FUNCTION mul$ (pri$, sec$)
DECLARE FUNCTION div$ (pri$, sec$)
DECLARE FUNCTION power$ (pri$, sec$)
DECLARE FUNCTION numero% (a$)
DECLARE FUNCTION bracket$ (e$, priorita%)
DECLARE FUNCTION neg$ (e$)
DECLARE FUNCTION der$ (e$)
DECLARE FUNCTION add$ (pri$, sec$)
DECLARE FUNCTION sub$ (pri$, sec$)

' crea abbastanza spazio sullo stack
CLEAR , , 6000

PRINT "Introduci la funzione f(x) da derivare"
PRINT : PRINT "f(x)    = ";
LINE INPUT fun$

' elimina gli spazi nella espressione
EliminaSpazi fun$

dfun$ = der$(LCASE$(fun$))
IF LEN(dfun$) THEN
   PRINT "D(f(x)) = "; dfun$
ELSE
   PRINT "ERRORE DI SINTASSI"
END IF

FUNCTION add$ (pri$, sec$)
      '===============================================================
      ' addizione   (pri + sec)
      '===============================================================
      IF pri$ = "0" THEN
              add$ = sec$
      ELSEIF sec$ = "0" THEN
              add$ = pri$
      ELSEIF numero%(pri$) AND numero%(sec$) THEN
              add$ = LTRIM$(RTRIM$(STR$(VAL(pri$) + VAL(sec$))))
      ELSEIF pri$ = sec$ THEN
              add$ = mul$("2", pri$)
      ELSEIF LEFT$(sec$, 1) = "-" THEN
              add$ = sub$(pri$, MID$(sec$, 2))
      ELSE
              add$ = pri$ + "+" + sec$
      END IF
END FUNCTION

FUNCTION bracket$ (e$, priorita)
      '===============================================================
      ' restituisce l'espressione racchiusa tra parentesi se l'operatore
      ' specificato ha priorità più alta dell'operatore principale
      ' dell'espressione ; la priorità è passata come numero negativo
      ' per l'operando di destra della espressione
```

```
    '================================================================
    DIM j, k, lun, parentesi
    bracket$ = e$
    lun = LEN(e$)

    IF LEFT$(e$, 1) = "-" THEN
        IF priorita < 0 THEN bracket$ = "(" + e$ + ")": EXIT FUNCTION
        k = 1
    END IF

    DO WHILE k < lun
        k = k + 1
        IF MID$(e$, k, 1) = "(" THEN
            parentesi = 1: j = k
            DO WHILE j < lun OR parentesi > 0
                j = j + 1
                parentesi = parentesi-(MID$(e$, j, 1) = "(") + _
                    (MID$(e$, j, 1) = ")")
            LOOP
            IF INSTR("^/*-+",MID$(e$,j,1))>ABS(priorita) THEN
                bracket$="("+e$+")"
            END IF
        END IF
    LOOP
END FUNCTION

FUNCTION der$ (e$)
    '================================================================
    ' Calcola la derivata di una espressione nella variabile X
    ' in caso di errore restituisce una stringa nulla
    '================================================================
    DIM pri$, sec$          'primo e secondo operando
    DIM dpri$, dsec$        'rispettive derivate
    DIM priorita            'priorita' massima degli operatori analizzati
    DIM posizione           'posizione dell'operatore con la massima priorit…
    DIM parentesi           'numero parentesi in sospeso
    DIM lun                 'lunghezza della stringa e$
    DIM k, j                'contatori e variabili temporanee

    ' ricerca l'operatore con la priorità massima
    priorita = 0: lun = LEN(e$)
    k = 0: IF LEFT$(e$, 1) = "-" THEN k = 1

    DO WHILE k < lun
        k = k + 1
        IF MID$(e$, k, 1) = ")" THEN der$ = "": EXIT FUNCTION
        IF MID$(e$, k, 1) = "(" THEN
            parentesi = 1: j = k
            DO WHILE j < lun
                j = j + 1
                parentesi = parentesi - (MID$(e$, j, 1) = "(") + (MID$(e$, j,
                    1) = ")")
                IF parentesi = 0 THEN EXIT DO
            LOOP
            IF parentesi THEN der$ = "": EXIT FUNCTION
            k = j
        END IF
        j = INSTR("^/*-+", MID$(e$, k, 1))
        IF j > priorita THEN priorita = j: posizione = k
    LOOP
```

```
' questo blocco tratta espressioni con due operandi
IF priorita > 0 THEN
        IF posizione = 1 OR posizione = lun THEN der$ = "": EXIT FUNCTION
        pri$ = LEFT$(e$, posizione - 1): sec$ = MID$(e$, posizione + 1)
        dpri$ = der$(pri$): dsec$ = der$(sec$)
        IF dpri$ = "" OR dsec$ = "" THEN der$ = "": EXIT FUNCTION
        SELECT CASE MID$(e$, posizione, 1)
                CASE "+": der$ = add$(dpri$, dsec$)
                CASE "-": der$ = sub$(dpri$, dsec$)
                CASE "*": der$ = add$(mul$(dpri$, sec$), mul$(pri$, dsec$))
                CASE "/": der$ = div$(sub$(mul$(dpri$, sec$), mul$(pri$, _
                        dsec$)), power$(sec$, "2"))
                CASE "^"
                IF dsec$ = "0" THEN
                        der$ = mul$(mul$(sec$, power$(pri$, sub$(sec$, "1"))), dpri$)
                ELSEIF dpri$ = "0" THEN
                        der$ = mul$(mul$(e$, "log(" + pri$ + ")"), dsec$)
                ELSE
                        der$ = mul$(e$, add$(mul$(dsec$, "log(" + pri$ + ")"), _
                                mul$(sec$, div$(dpri$, pri$))))
                END IF
        END SELECT
        EXIT FUNCTION
END IF

' se l'espressione ha un solo operando preceduto dal segno meno
IF LEFT$(e$, 1) = "-" THEN
        dpri$ = der$(MID$(e$, 2))
        IF dpri$ <> "" THEN der$ = neg$(dpri$) ELSE der$ = ""
        EXIT FUNCTION
END IF

'se è una potenza della variabile X
IF LEFT$(e$, 1) = "x" THEN
        IF e$ = "x" THEN der$ = "1": EXIT FUNCTION
        pri$ = MID$(e$, 2)
        IF numero%(pri$) THEN
                der$ = mul$(pri$, power$("x", LTRIM$(RTRIM$(STR$(VAL(pri$) _
                        - 1)))))
                EXIT FUNCTION
        ELSE
                der$ = "": EXIT FUNCTION
        END IF
END IF

' semplifica se si tratta du un unico operando racchiuso tra parentesi
IF LEFT$(e$, 1) = "(" AND RIGHT$(e$, 1) = ")" THEN
        der$ = der$(MID$(e$, 2, lun - 2)): EXIT FUNCTION
END IF


' se non vi sono parentesi pu• trattarsi solo di una costante
IF RIGHT$(e$, 1) <> ")" THEN
        IF INSTR(e$, "x") = 0 THEN der$ = "0" ELSE der$ = ""
        EXIT FUNCTION
END IF

'in tutti gli altri casi deve trattarsi di una funzione
k = INSTR(e$, "(")
```

```
        IF k = 0 OR k > 5 OR k = lu - 1 THEN der$ = "": EXIT FUNCTION

        pri$ = LEFT$(e$, k - 1)              'il nome della funzione
        sec$ = MID$(e$, k + 1, lun - k - 1)
        dsec$ = der$(sec$): IF dsec$ = "" THEN der$ = "": EXIT FUNCTION

        SELECT CASE pri$
                CASE "sin": der$ = mul$("cos(" + sec$ + ")", dsec$)
                CASE "cos": der$ = mul$("-sin(" + sec$ + ")", dsec$)
                CASE "tan": der$ = div$(dsec$, "cos(" + sec$ + ")^2")
                CASE "log": der$ = div$(dsec$, sec$)
                CASE "exp": der$ = mul$("exp(" + sec$ + ")", dsec$)
                CASE "sqr": der$ = div$(div$(dsec$, "2"), power$(sec$, ".5"))
                CASE ELSE: der$ = ""
        END SELECT
END FUNCTION

SUB EliminaSpazi (a$)
        '________________________________
        ' elimina gli spazi in una stringa
        '________________________________
        DIM i
        DO
                i = INSTR(i+1, a$, " ")
                IF i THEN a$ = LEFT$(a$, i - 1) + MID$(a$, i + 1)
        LOOP WHILE i
END SUB

FUNCTION div$ (pri$, sec$)
        '===============================================================
        ' divisione    (pri / sec)
        '===============================================================
        IF pri$ = "0" THEN
                div$ = "0"
        ELSEIF sec$ = "1" THEN
                div$ = pri$
        ELSEIF sec$ = "-1" THEN
                div$ = neg$(pri$)
        ELSEIF pri$ = sec$ THEN
                div$ = "1"
        ELSEIF pri$ = neg$(sec$) THEN
                div$ = "-1"
        ELSEIF numero%(pri$) AND numero%(sec$) THEN
                IF VAL(sec$) = 0 THEN
                        div$ = ""
                ELSEIF VAL(pri$) / VAL(sec$) = VAL(pri$) \ VAL(sec$) THEN
                        div$ = LTRIM$(RTRIM$(STR$(VAL(pri$) / VAL(sec$))))
                ELSE
                        div$ = bracket$(pri$, 3) + "/" + bracket$(sec$, -3)
                END IF
        ELSE
                div$ = bracket$(pri$, 3) + "/" + bracket$(sec$, -3)
        END IF
END FUNCTION

FUNCTION mul$ (pri$, sec$)
        '===============================================================
        ' moltiplicazione    (pri * sec)
        '===============================================================
        IF pri$ = "0" OR sec$ = "0" THEN
```

```
        mul$ = "0"
    ELSEIF pri$ = "1" THEN
        mul$ = sec$
    ELSEIF sec$ = "1" THEN
        mul$ = pri$
    ELSEIF pri$ = "-1" THEN
        mul$ = neg$(sec$)
    ELSEIF sec$ = "-1" THEN
        mul$ = neg$(pri$)
    ELSEIF numero%(pri$) AND numero%(sec$) THEN
        mul$ = LTRIM$(RTRIM$(STR$(VAL(pri$) * VAL(sec$))))
    ELSE
        mul$ = bracket$(pri$, 3) + "*" + bracket$(sec$, -3)
    END IF
END FUNCTION

FUNCTION neg$ (e$)
    '================================================================
    ' cambio segno    ( -pri )
    '================================================================
    IF numero%(e$) THEN
            neg$ = LTRIM$(RTRIM$(STR$(-VAL(e$))))
    ELSE
            a$ = bracket$(e$, 3)
            IF LEFT$(a$, 1) = "-" THEN neg$ = MID$(a$, 2) ELSE neg$ = "-" + a$
    END IF
END FUNCTION


FUNCTION numero% (a$)
    '================================================================
    ' determina se l'argomento è un numero
    '================================================================
    DIM i, decFlg
    numero% = -1                        ' assume VERO
    decFlg = 0                          ' questo flag diventa TRUE quando
                                        ' incontriamo il punto decimale
    FOR i = 1 TO LEN(a$)
            SELECT CASE MID$(a$, i, 1)
                    CASE "0" TO "9"
                    CASE "+", "-"
                            IF i <> 1 THEN numero% = 0: EXIT FOR
                    CASE "."
                            IF decFlg THEN numero% = 0: EXIT FOR
                            decFlg = -1
                    CASE ELSE
                            numero% = 0: EXIT FOR
            END SELECT
    NEXT i
END FUNCTION

FUNCTION power$ (pri$, sec$)
    '================================================================
    ' elevamento a potenza  (pri ^ sec)
    '================================================================
    IF sec$ = "0" THEN
            power$ = "1"
    ELSEIF sec$ = "1" THEN
            power$ = pri$
    ELSEIF numero%(pri$) AND numero%(sec$) THEN
            power$ = LTRIM$(RTRIM$(STR$(VAL(pri$) ^ VAL(sec$))))
```

```
        ELSEIF VAL(sec$) = .5 THEN
                power$ = "sqr(" + pri$ + ")"
        ELSEIF pri$ = "x" AND numero%(sec$) AND VAL(sec$)=ABS(INT(VAL(sec$))) THEN
                power$ = "x" + sec$
        ELSE
                power$ = bracket$(pri$, 5) + "^" + bracket$(sec$, -5)
        END IF
END FUNCTION

FUNCTION sub$ (pri$, sec$)
        '================================================================
        ' sottrazione  (pri - sec)
        '================================================================
        IF pri$ = "0" THEN
                sub$ = neg$(sec$)
        ELSEIF sec$ = "0" THEN
                sub$ = pri$
        ELSEIF pri$ = sec$ THEN
                sub$ = "0"
        ELSEIF numero%(pri$) AND numero%(sec$) THEN
                sub$ = LTRIM$(RTRIM$(STR$(VAL(pri$) - VAL(sec$))))
        ELSEIF LEFT$(sec$, 1) = "-" THEN
                sub$ = add$(pri$, MID$(sec$, 2))
        ELSE
                sub$ = pri$ + "-" + sec$
        END IF
END FUNCTION
```

EVAL, VALUTATORE DI ESPRESSIONI MATEMATICHE

Vi sono occasioni in cui sarebbe molto utile poter valutare una espressione matematica introdotta dall'utente, ad esempio in un programma scientifico per tracciare una funzione, oppure in alcune applicazioni gestionali per ricercare dati in un database usando un criterio di ricerca complesso, o ancora per calcolare dei trend statistici dai dati memorizzati su file. Naturalmente, un *parser* di espressioni è anche alla base di tutti i compilatori ed interpreti, tra cui il mio compilatore per file batch BATCH WIZARD e il generatore di sistemi esperti EXPERTO, ambedue distribuiti da SoftWhale.

A prima vista, costruire un valutatore di espressioni sembra un compito non troppo difficile; ben presto ci si rende conto di dover tenere traccia di parentesi nidificate, di funzioni con uno, due o più argomenti, per non parlare di alcune possibili ambiguità da risolvere (ad esempio, il segno "-" che può essere sia un operatore unario che binario). Molti dei problemi derivano dalla necessità di ritardare la valutazione di una operazione fino a quando sono stati letti tutti i suoi operandi; consideriamo la seguente espressione

```
(1)            12 * (34 + 56)
```

o la più complessa

```
(2)            EXP(2 * SIN(0.5) + 1 / COS(0.5)) ^ 2
```

Quando incontriamo il nome di una funzione o il simbolo di un operatore dobbiamo memorizzarlo da qualche parte per poterlo eseguire solo in seguito; anche prima di spiegare nel dettaglio come funziona effettivamente l'algoritmo,

è chiaro che la maggior parte del tempo è speso nell'analisi (*parsing*) dell'espressione, piuttosto che nella sua valutazione. ì

LA NOTAZIONE POLACCA INVERSA

Un sistema per evitare quasi completamente il problema del *parsing* è di introdurre l'espressione da valutare nella cosiddetta *Notazione Polacca Inversa* (spesso abbreviata in RPN, da *Reverse Polish Notation*), conosciuta anche come notazione postfissa, in cui gli operatori seguono gli operandi a cui si riferiscono; questo tipo di notazione è diventata abbastanza conosciuta con la diffusione delle calcolatrici della Hewlett-Packard, quindi è possibile che alcuni di voi sappiano già di cosa si tratti.

Riassumendo in poche parole, in una calcolatrice RPN gli operandi e gli operatori devono essere introdotti nell'ordine in cui devono essere applicati; ad esempio, l'espressione (1) dovrebbe essere introdotta come

```
(1*)            12 <enter> 34 <enter> 56 + *
```

dove il tasto <enter> è necessario per separare un numero dal successivo; è evidente che le espressioni RPN non richiedono mai le parentesi, e che non vi è alcun bisogno di tenere conto delle priorità dei vari operatori, che è uno dei compiti più ardui di un parser: ad esempio, 2^3^4 deve essere calcolato come (2^3)^4 o come 2^(3^4) ? D'altra parte, mentre le espressioni RPN sono in genere più corte delle espressioni algebriche corrispondenti, esse tendono ad essere abbastanza criptiche, soprattutto quando includono più di due o tre operatori; ecco la traduzione in RPN della espressione (2) vista in precedenza

```
(2*)            2 <enter> 0.5 SIN * 1 <enter> 0.5 COS / + EXP 2 ^
```

Se vi state chiedendo perché un ingegnere o una scienziato avrebbe dovuto preferire le calcolatrici RPN ed usare una notazione così astrusa, la risposta è una soltanto: la velocità di esecuzione! Infatti, le calcolatrici RPN erano (e sono tuttora) molto più veloci (fino a 5-6 volte) delle calcolatrici ordinarie ("algebriche"). Inoltre la notazione RPN è decisamente più compatta di quella algebrica, e fino a non molti anni fa la calcolatrice più potente disponeva di poco poco più di 2K di memoria RAM. Persino sui personal computer attuali, i linguaggi orientati alla notazione polacca inversa come il FORTH godono della meritata reputazione di produrre il codice più compatto e veloce di qualsiasi altro linguaggio di programmazione, C incluso. Come passo intermedio nella costruzione di un completo valutatore di espressioni, vi mostrerò una piccola routine che valuta una qualsiasi espressione RPN passata ad essa come argomento

```
DEFINT A-Z
FUNCTION EvalRPN# (rpnExpr$)
```

```
'==================================================================
' Un semplice valutatore di espressioni RPN
'
' Supporta solo i numeri positivi e le quattro operazioni
' Errori di sintassi ed espressioni non bilanciate restituiscono zero
'==================================================================
DIM index, temp, sp, stak(50) AS DOUBLE

DO
   index = index + 1

   SELECT CASE MID$(rpnExpr$, index, 1)
      CASE "0" TO "9", "."
         temp = index
         DO
            index = index + 1
         LOOP WHILE INSTR("0123456789.", MID$(rpnExpr$, index, 1))
         sp = sp + 1
         stak(sp) = VAL(MID$(rpnExpr$, temp, index - temp))
      CASE "*"
         stak(sp - 1) = stak(sp - 1) * stak(sp)
         sp = sp - 1
      CASE "/"
         stak(sp - 1) = stak(sp - 1) / stak(sp)
         sp = sp - 1
      CASE "+"
         stak(sp - 1) = stak(sp - 1) + stak(sp)
         sp = sp - 1
      CASE "-"
         stak(sp - 1) = stak(sp - 1) - stak(sp)
         sp = sp - 1
      CASE " "
      CASE ""
         ' fine della espressione
         IF sp = 1 THEN EvalRPN = stak(sp)
         EXIT DO
      CASE ELSE
         EXIT DO
   END SELECT
LOOP
END FUNCTION
```

Poiché questa è una routine demo ho preferito non complicarla troppo, e infatti essa prevede solo costanti positive e le quattro operazioni; tenete conto però che si tratta di appena 50 righe di codice! Per testarla e vederla al lavoro, provate ad eseguire in modalità trace i seguenti comandi

```
PRINT EvalRPN("12 34 46+*")
PRINT EvalRPN("12 34+46 76*/")
```

notate che gli spazi sono necessari per separare due operandi consecutivi, un po' come il tasto <invio> sulle calcolatrici RPN.

IL PROGRAMMA EVAL.BAS

E' evidente che riuscendo a trovare un metodo per trasformare una espressione algebrica nella sua equivalente RPN sarebbe molto semplice valutarla. Ma

c'è un aspetto della questione ancora più interessante: se riuscissimo a convertire in RPN una espressione contenente una o più variabili, essa potrebbe essere eseguita per valori differenti della variabile senza dover eseguire nuovamente la conversione ogni volta, e risparmiando quindi una notevole quantità di tempo di caloclo. Difatti, questo è quello che fa il compilatore BC.EXE quando converte una espressione BASIC in codice macchina: dopo che l'espressione è stata compilata (in una forma simile alla RPN) essa viene eseguita molto velocemente, ed inoltre il programma compilato non deve necessariamente includere il *parser* delle espressioni.

Per questo motivo ho deciso di suddividere il valutatore di espressioni in cinque routine differenti, in modo da poter separare nettamente il processo di parsing e conversione (o "compilazione" della espressione) dalla valutazione vera e propria; in tal modo è anche possibile distinguere più facilmente gli errori in fase di parsing (ad esempio una espressione non bilanciata) dagli errori a runtime (ad esempio, la divisione per zero o il logaritmo di un numero negativo). Le routine che compongono il pacchetto sono le seguenti:

CompileExpression$ converte una espressione algebrica nel formato RPN

EvalExpression# valuta una espressione "compilata" in RPN

EvalError restituisce il più recente errore di compilazione o runtime

DeclareVariable crea una variabile che può essere usata nelle espressioni

SetVariable assegna un valore ad una variabile creata in precedenza

Ecco il listato completo del valutatore di espressioni:

```
'===========================================================================
'
'                       VALUTATORE DI ESPRESSIONI
'
'                 (C) 1992-94 Francesco Balena / SoftWhale
'
'===========================================================================
'$INCLUDE: 'eval.bi'
DEFINT A-Z

CONST VARMAX = 50                    ' massimo numero di variabili
CONST STACKMAX = 20                  ' massimo numero di operatori in sospeso

'-------- OPCODE  TABLE ---------------

CONST opOPENBRACKET = -1             ' (questi codici non sono mai inseriti
CONST opCLOSEBRACKET = -2            '  nelle espressioni compilate)
CONST opCOMMA = -3                   '

' ---- SPECIAL OPCODES ----
CONST opEND = 0                      ' fine della espressione
```

```
CONST opNUM = opEND + 1              ' costanti numeriche - seguito da MKD$(number)
CONST opVAR = opNUM + 1              ' variabili - seguito da MKI$(varIndex)


' —— UNARY OPCODES ——
CONST opMINUS = opVAR + 1            ' meno unario
CONST opNOT = opMINUS + 1            ' NOT booleano
CONST opABS = opNOT + 1              ' (prima funzione ad un operando)
CONST opINT = opABS + 1              '
CONST opFIX = opINT + 1              '
CONST opSGN = opFIX + 1              '
CONST opSQR = opSGN + 1              '
CONST opLOG = opSQR + 1              '
CONST opEXP = opLOG + 1              '
CONST opSIN = opEXP + 1              '
CONST opCOS = opSIN + 1              '
CONST opTAN = opCOS + 1              '
CONST opATN = opTAN + 1              '


CONST opTWOARGS = opATN + 1          ' —— CODICI BINARI ——
CONST opMIN = opTWOARGS              ' (prima funzione a due operandi)
CONST opMAX = opMIN + 1              '
CONST opPOWER = opMAX + 1            ' elevamento a potenza
CONST opMUL = opPOWER + 1            ' moltiplicazione
CONST opDIV = opMUL + 1              ' divisione
CONST opINTDIV = opDIV + 1           ' divisione intera
CONST opMOD = opINTDIV + 1           ' modulo
CONST opADD = opMOD + 1              ' addizione
CONST opSUB = opADD + 1              ' sottrazione
CONST opEQ = opSUB + 1               ' uguale a        ( booleano )
CONST opLT = opEQ + 1                ' minore di
CONST opLE = opLT + 1                ' minore o uguale a
CONST opGT = opLE + 1                ' maggiore di
CONST opGE = opGT + 1                ' maggiore o uguale a
CONST opNE = opGE + 1                ' non uguale a
CONST opAND = opNE + 1               ' AND booleano
CONST opOR = opAND + 1               ' OR booleano
CONST opXOR = opOR + 1               ' XOR booleano


CONST opTHREEARGS = opXOR + 1        ' —— TRE OPERANDI ——
CONST opIFF = opTHREEARGS            ' (prima funzione a tre operandi)

CONST opFOURARGS = opIFF + 1         ' —— QUATTRO OPERANDI ——


'———— VARIABILI CONDIVISE ————————

DIM SHARED VarName$(VARMAX)
DIM SHARED VarValue(VARMAX) AS DOUBLE
DIM SHARED SharedErrorCode


ON ERROR GOTO GetErrorCode


'____________________________________
' Routine per il trattamento degli errori
' (vedere nota all'inizio della routine EvalExpression)
```

```
'________________________________

GetErrorCode:
      SharedErrorCode = ERR
      RESUME NEXT

FUNCTION CompileExpression$ (expression$)

'=============================================================
'   Compila una espressione algebrica in RPN
'
'   Gli errori possono essere testati con la funzione EvalError
'=============================================================

DIM index, sp, opSp, waitForOperator, sign, varIndex
DIM expr$, temp, tmp$, digits$, compiled$
DIM opStack(STACKMAX), prStack(STACKMAX), argStack(STACKMAX)

'______________________________
'   Inizializzazione
'______________________________

SharedErrorCode = 0                    ' cancella il codice di errore
digits$ = "0123456789"                 ' una costante stringa utile
prStack(0) = 255                       ' questa è la max. priorità
argStack(0) = 1                        ' ci aspettiamo una espressione

sign = 1
expr$ = UCASE$(expression$) + " "
compiled$ = ""
index = 1

'______________________________
'   Ciclo principale
'______________________________

DO
      GOSUB CompileExpressionSkipBlanks

      IF waitForOperator = 0 THEN

            SELECT CASE MID$(expr$, index, 1)

            CASE "0" TO "9", "."                   ' costanti numeriche
                  temp = index
                  GOSUB CompileExpressionSkipDigits
                  IF MID$(expr$, index, 1) = "." THEN GOSUB CompileExpressionSkipDigits
                  ' skip the exponent, if any
                  IF INSTR("ED", MID$(expr$, index, 1)) THEN
                        index = index + 1
                        IF INSTR("+-", MID$(expr$, index, 1)) THEN index = index + 1
                        GOSUB CompileExpressionSkipDigits
                  END IF
                  ' trattamento dei numeri negativi
                  ' push del numero sullo stack, aggiorna compiled$
                  sp = sp + 1
                  compiled$ = compiled$ + CHR$(1) + MKD$(sign * VAL(MID$(expr$, temp, _
                        index - temp)))
                  sign = 1
                  waitForOperator = -1
```

```
CASE "+"                                    ' più unario
        index = index + 1                   ' è semplicemente ignorato


CASE "-"                        ' meno unario
        ' se precede una cifra, ricorda che il numero che segue è negativo
        ' altrimenti si tratta di un meno unario
        index = index + 1
        GOSUB CompileExpressionSkipBlanks
        IF INSTR(digits$, MID$(expr$, index, 1)) THEN
                sign = -1
        ELSE
                opSp = opSp + 1
                opStack(opSp) = opMINUS
                prStack(opSp) = 18
                argStack(opSp) = 1
        END IF

CASE "("
        opSp = opSp + 1
        opStack(opSp) = opOPENBRACKET
        prStack(opSp) = 255
        argStack(opSp) = argStack(opSp - 1)
        index = index + 1

CASE "A" TO "Z"
        ' potrebbe essere il nome di una funzione o di una variabile
        ' oppure il NOT booleano
        temp = index
        GOSUB CompileExpressionSkipAlphanum
        IF index < LEN(expr$) THEN GOSUB CompileExpressionSkipBlanks
        tmp$ = RTRIM$(MID$(expr$, temp, index - temp))

        IF tmp$ = "NOT" THEN
                ' trattamento separato del NOT booleano
                opSp = opSp + 1
                opStack(opSp) = opNOT
                prStack(opSp) = 8
                argStack(opSp) = 1

        ELSEIF MID$(expr$, index, 1) <> "(" THEN
                ' se non è seguita da "(" deve essere il nome di una variabile
                varIndex = 0
                FOR temp = 1 TO UBOUND(VarName$)
                        IF VarName$(temp) = tmp$ THEN varIndex = temp: EXIT FOR
                NEXT
                IF varIndex = 0 THEN SharedErrorCode = -3: EXIT FUNCTION
                sp = sp + 1
                compiled$ = compiled$ + CHR$(opVAR) + MKI$(varIndex)
                waitForOperator = -1

        ELSE
                ' se è il nome di una funzione
                ' prepara ad eseguire il push di un opcode sullo stack
                opSp = opSp + 1
                ' assume this is a one-operand function
                argStack(opSp) = 1
                ' all functions have the same priority
                prStack(opSp) = 30
```

```
                    SELECT CASE tmp$
                        CASE "ABS"
                            opStack(opSp) = opABS
                        CASE "INT"
                            opStack(opSp) = opINT
                        CASE "FIX"
                            opStack(opSp) = opFIX
                        CASE "SGN"
                            opStack(opSp) = opSGN
                        CASE "SQR"
                            opStack(opSp) = opSQR
                        CASE "LOG"
                            opStack(opSp) = opLOG
                        CASE "EXP"
                            opStack(opSp) = opEXP
                        CASE "SIN"
                            opStack(opSp) = opSIN
                        CASE "COS"
                            opStack(opSp) = opCOS
                        CASE "TAN"
                            opStack(opSp) = opTAN
                        CASE "ATN"
                            opStack(opSp) = opATN
                        CASE "MIN"
                            opStack(opSp) = opMIN
                            argStack(opSp) = 2
                        CASE "MAX"
                            opStack(opSp) = opMAX
                            argStack(opSp) = 2
                        CASE "IFF"
                            opStack(opSp) = opIFF
                            argStack(opSp) = 3

                        '**** aggiungere qui le nuove funzioni ****

                        CASE ELSE
                            SharedErrorCode = -2: EXIT FUNCTION
                    END SELECT

                END IF

            CASE ELSE
                GOTO CompileExpressionSyntaxError

            END SELECT

        ELSE

            '________________________________
            '   stiamo aspettando un operatore
            '________________________________

            SELECT CASE MID$(expr$, index, 1)


            CASE ""                                 ' fine della espressione
                currOpcode = opEND
                currPriority = 0
            CASE ")"
                currOpcode = opCLOSEBRACKET
```

```
        currPriority = 0
CASE ","
        currOpcode = opCOMMA
        currPriority = 0
CASE "^"
        currOpcode = opPOWER
        currPriority = 20
CASE "*"
        currOpcode = opMUL
        currPriority = 17
CASE "/"
        currOpcode = opDIV
        currPriority = 17
CASE "\"                                ' divisione intera
        currOpcode = opINTDIV
        currPriority = 16
CASE "+"
        currOpcode = opADD
        currPriority = 14
CASE "-"
        currOpcode = opSUB
        currPriority = 14
CASE "="
        currOpcode = opEQ
        currPriority = 10
CASE "<"
        IF MID$(expr$, index + 1, 1) = "=" THEN
                currOpcode = opLE
                index = index + 1
        ELSEIF MID$(expr$, index + 1, 1) = ">" THEN
                currOpcode = opNE
                index = index + 1
        ELSE
                currOpcode = opLT
        END IF
        currPriority = 10
CASE ">"
        IF MID$(expr$, index + 1, 1) = "=" THEN
                currOpcode = opGE
                index = index + 1
        ELSE
                currOpcode = opGT
        END IF
        currPriority = 10
CASE ELSE
        ' se gli altri test sono falliti ci rimane da testare
        ' se si tratta di un operatore AND, OR, XOR o MOD
        ' questi devono essere preceduti da uno spazio o ")"
        IF INSTR(" )", MID$(expr$, index - 1, 1)) = 0 THEN
                GOTO CompileExpressionSyntaxError
        END IF
        temp = index
        GOSUB CompileExpressionSkipAlphanum
        tmp$ = MID$(expr$, temp, index - temp)


        SELECT CASE tmp$
                CASE "MOD"
                        currOpcode = opMOD
                        currPriority = 15
```

```
                CASE "AND"
                        currOpcode = opAND
                        currPriority = 7
                CASE "OR"
                        currOpcode = opOR
                        currPriority = 6
                CASE "XOR"
                        currOpcode = opXOR
                        currPriority = 6
                CASE ELSE
                        GOTO CompileExpressionSyntaxError
        END SELECT

        ' questi operatori devono essere seguiti da uno spazio o "("
        IF INSTR(" (", MID$(expr$, index, 1)) = 0 THEN
                GOTO CompileExpressionSyntaxError
        END IF

        ' torna indietro di un carattere, in modo che la routine
        ' Wait-For-Operand possa processare correttamente la parentesi
        index = index - 1
END SELECT

' salta l'operatore appena processato
index = index + 1

' se non è un ")" dobbiamo prepararci a leggere un operando
IF currOpcode <> opCLOSEBRACKET THEN waitForOperator = 0

'_________________________________
' Questa sezione della routine confronta la priorità dell'operatore
' appena analizzato con la priorità degli altri operatori in sospeso
' in cima allo stack opStack()
' Fin quando la priorità dell'operatore corrente o della funzione
' è maggiore della priorità dell'operatore o funzione in cima a
' opStack(), quest'ultimo deve essere prelevato dallo stack ed
' "eseguito"; 255 è la priorità massima, ed è assegnata a "(" e allo
' stato fittizio "inizio-espressione"; il test nel ciclo DO-WHILE
' impedisce di eseguire il pop di troppi elementi da opStack().
'_________________________________

DO WHILE currPriority <= prStack(opSp) AND prStack(opSp) <> 255
        compiled$ = compiled$ + CHR$(opStack(opSp))
        IF opStack(opSp) >= opFOURARGS THEN
                ' funzione a quattro operandi
                sp = sp - 3
        ELSEIF opStack(opSp) >= opTHREEARGS THEN
                ' funzione a tre operandi
                sp = sp - 2
        ELSEIF opStack(opSp) >= opTWOARGS THEN
                ' funzione a due operandi o operatore binario (+ - / *)
                sp = sp - 1
        END IF
        ' preleva l'operatore dallo stack (pop)
        opSp = opSp - 1
        ' controlla che l'espressione sia ben bilanciata
        IF opSp < 0 OR sp <= 0 THEN GOTO CompileExpressionSyntaxError
LOOP

'_________________________________
'  Alcuni operatori richiedono un trattamento speciale
```

```
                '_____________________________________

                SELECT CASE currOpcode
                        CASE opEND
                                IF opSp THEN GOTO CompileExpressionSyntaxError
                                IF argStack(opSp) <> 1 THEN GOTO CompileExpressionWrongArgs
                                CompileExpression$ = compiled$ + CHR$(opEND)
                                EXIT FUNCTION
                        CASE opCLOSEBRACKET
                                IF opStack(opSp) <> opOPENBRACKET THEN GOTO
CompileExpressionSyntaxError
                                opSp = opSp - 1                 ' scarta la "(" dallo stack
                        CASE opCOMMA
                                argStack(opSp) = argStack(opSp) - 1
                                IF argStack(opSp) = 0 THEN GOTO CompileExpressionWrongArgs
                                waitForOperator = 0
                        CASE ELSE
                                ' tutti i rimanenti codici devono essere inseriti sullo stack
(push)
                                opSp = opSp + 1
                                opStack(opSp) = currOpcode
                                prStack(opSp) = currPriority
                                argStack(opSp) = argStack(opSp - 1)
                END SELECT
        END IF

LOOP

' questa linea non è mai eseguita ...

'_________________________________
'   SUBROUTINE
'_________________________________

CompileExpressionSkipBlanks:
        ' salta tutti gli spazi
        DO WHILE MID$(expr$, index, 1) = " "
                index = index + 1
        LOOP
        RETURN

CompileExpressionSkipDigits:
        ' sposta il puntatore dopo un numero
        IF INSTR(digits$, MID$(expr$, index, 1)) = 0 THEN
                IF INSTR(digits$, MID$(expr$, index + 1, 1)) = 0 THEN
                        GOTO CompileExpressionSyntaxError
                END IF
        END IF
        DO
                index = index + 1
        LOOP WHILE INSTR(digits$, MID$(expr$, index, 1))
        RETURN

CompileExpressionSkipAlphanum:
        ' sposta il puntatore dopo una stringa di lettere e cifre
        DO WHILE INSTR("ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789", UCASE$(MID$(expr$, index, 1)))
                index = index + 1
        LOOP
        RETURN
```

```
CompileExpressionGetNumber:
      GOSUB CompileExpressionSkipDigits
      IF MID$(expr$, index, 1) = "." THEN GOSUB CompileExpressionSkipDigits
      ' salta l'esponente, se esiste
      IF INSTR("ED", MID$(expr$, index, 1)) THEN
              index = index + 1
              IF INSTR("+-", MID$(expr$, index, 1)) THEN index = index + 1
              GOSUB CompileExpressionSkipDigits
      END IF
      ' carica un numero sullo stack, e aggiorna compiled$
      sp = sp + 1
      compiled$ = compiled$ + CHR$(1) + MKD$(VAL(MID$(expr$, temp, index - temp)))
      RETURN

CompileExpressionSyntaxError:
      SharedErrorCode = -1
      EXIT FUNCTION

CompileExpressionWrongArgs:
      SharedErrorCode = -4
      EXIT FUNCTION

END FUNCTION

SUB DeclareVariable (index, vname$)

      '================================================================
      '  Dichiara o cancella una variabile
      '
      '  se VNAME$<>"" INDEX è restituito pari alla posizione
      '  della variabile appena creata nell'array varname$()
      '================================================================

      IF LEN(vname$) THEN
              DO
                      index = index + 1
              LOOP WHILE LEN(VarName$(index)) AND VarName$(index) <> UCASE$(vname$)
      END IF

      VarName$(index) = UCASE$(vname$)

END SUB

FUNCTION EvalError

      '==================================================================
      '    Restituisce l'errore di compilazione o runtime più recente
      '
      '   Codici di errore in fase di compilazione :
      '      -1   errore di sintassi
      '      -2   funzione non definita
      '      -3   variabile non definita
      '      -4   numero errato di argomenti
      '
      '   I codici di errore a runtime sono gli stessi del BASIC
      '
      '==================================================================

      EvalError = SharedErrorCode
```

```
END FUNCTION

FUNCTION EvalExpression# (compiled$)

'================================================================
'   Valuta una espressione compilata
'
'   Gli errori sono restituiti dalla funzione EvalError
'================================================================

' NOTA: se il programma è eseguito sotto BASIC PDS 7.x o VBDOS
'       si può eliminare il commento alla riga seguente e cancellare
'       la routine di trattamento errori nel modulo principale

'ON LOCAL ERROR GOTO EvalExpressionError

DIM index, sp
DIM Tos AS DOUBLE, stak(STACKMAX) AS DOUBLE

sp = -1

DO
      ' ciclo principale ------------------


      index = index + 1
      SELECT CASE ASC(MID$(compiled$, index))
              CASE opEND
                      EvalExpression = Tos
                      EXIT FUNCTION
              CASE opNUM
                      sp = sp + 1
                      stak(sp) = Tos
                      Tos = CVD(MID$(compiled$, index + 1, 8))
                      index = index + 8
              CASE opVAR
                      sp = sp + 1
                      stak(sp) = Tos
                      Tos = VarValue(CVI(MID$(compiled$, index + 1, 2)))
                      index = index + 2
              CASE opMINUS
                      Tos = -Tos
              CASE opNOT
                      Tos = NOT Tos
              CASE opABS
                      Tos = ABS(Tos)
              CASE opINT
                      Tos = INT(Tos)
              CASE opFIX
                      Tos = FIX(Tos)
              CASE opSGN
                      Tos = SGN(Tos)
              CASE opSQR
                      Tos = SQR(Tos)
              CASE opLOG
                      Tos = LOG(Tos)
              CASE opEXP
                      Tos = EXP(Tos)
              CASE opSIN
                      Tos = SIN(Tos)
```

```
CASE opCOS
        Tos = COS(Tos)
CASE opTAN
        Tos = TAN(Tos)
CASE opATN
        Tos = ATN(Tos)
CASE opPOWER
        Tos = stak(sp) ^ Tos
        sp = sp - 1
CASE opMUL
        Tos = stak(sp) * Tos
        sp = sp - 1
CASE opDIV
        Tos = stak(sp) / Tos
        sp = sp - 1
CASE opINTDIV
        Tos = stak(sp) \ Tos
        sp = sp - 1
CASE opMOD
        Tos = stak(sp) MOD Tos
        sp = sp - 1
CASE opADD
        Tos = stak(sp) + Tos
        sp = sp - 1
CASE opSUB
        Tos = stak(sp) - Tos
        sp = sp - 1
CASE opEQ
        Tos = stak(sp) = Tos
        sp = sp - 1
CASE opLT
        Tos = stak(sp) < Tos
        sp = sp - 1
CASE opLE
        Tos = stak(sp) <= Tos
        sp = sp - 1
CASE opGT
        Tos = stak(sp) > Tos
        sp = sp - 1
CASE opGE
        Tos = stak(sp) >= Tos
        sp = sp - 1
CASE opNE
        Tos = stak(sp) <> Tos
        sp = sp - 1
CASE opAND
        Tos = stak(sp) AND Tos
        sp = sp - 1
CASE opOR
        Tos = stak(sp) OR Tos
        sp = sp - 1
CASE opXOR
        Tos = stak(sp) XOR Tos
        sp = sp - 1
CASE opMIN
        IF stak(sp) < Tos THEN Tos = stak(sp)
        sp = sp - 1
CASE opMAX
        IF stak(sp) > Tos THEN Tos = stak(sp)
        sp = sp - 1
```

```
            CASE opIFF
                    IF stak(sp - 1) THEN Tos = stak(sp)
                    sp = sp - 2

            '**** aggiungere qui i nuovi codici ****

        END SELECT

        IF SharedErrorCode THEN EXIT FUNCTION
LOOP

' questa linea non è mai eseguita ...


' routine locale per il trattamento degli errori
' vedere nota all'inizio di questa procedura

'EvalExpressionError:
'       SharedErrorCode = ERR
'       RESUME NEXT

END FUNCTION

SUB SetVariable (index, value#)

        '================================================================
        '   Assegna un nuovo valore ad una variabile esistente
        '================================================================

        VarValue#(index) = value#

END SUB
```

Le procedure condividono alcune variabili e sono state progettare per essere contenute in un modulo separato o in una Quick Library; infatti, le tre procedure **EvalError**, **DeclareVariable** e **SetVariable** non sono che delle "access procedure" che permettono ad altri moduli del programma di accedere a queste strutture e variabili interne. Per usare le procedure in un programma dovrete includere la seguente direttiva

```
'$INCLUDE: 'eval.bi'
```

all'inizio di tutti i moduli che devono accedere al valutatore di espressioni. Questo è il contenuto del file di include

```
'   EVAL.BI : File di include per accedere alle routine in EVAL.BAS
DECLARE SUB DeclareVariable (index%, vname$)
DECLARE SUB SetVariable (index%, value#)
DECLARE FUNCTION CompileExpression$ (expression$)
DECLARE FUNCTION EvalError% ()
DECLARE FUNCTION EvalExpression# (compiled$)
```

E' molto semplice valutare una espressione introdotta durante l'esecuzione dall'operatore

```
LINE INPUT "Introdurre una espressione: "; expression$
compiled$ = CompileExpression$(expression$)
IF EvalError then PRINT "Errore di compilazione": END
result# = EvalExpression# (compiled$)
IF EvalError then PRINT "Errore aritmetico": END
PRINT "Risultato = "; result#
```

Naturalmente è possibile (e preferibile) fornire più dettagli in caso di errore; notate che tutti gli errori di compilazione (errori sintattici) sono negativi, mentre tutti gli errori a runtime (errori aritmetici) sono positivi e seguono la stessa numerazione del BASIC

LINE INPUT "Introdurre una espressione: "; expression$

compiled$ = CompileExpression$(expression$)

```
SELECT CASE EvalError
      CASE -1: PRINT "Errore di sintassi": END
      CASE -2: PRINT "Funzione non definita": END
      CASE -3: PRINT "Variabile non definita": END
      CASE -4: PRINT "Numero errato di argomenti": END
END SELECT
result# = EvalExpression# (compiled$)
SELECT CASE EvalError
      CASE 5: PRINT "Chiamata illegale a funzione": END
      CASE 6: PRINT "Overflow": END
      CASE 11: PRINT "Divisione per zero": END
      CASE ELSE: PRINT "Runtime Error": END
END SELECT
```

Il programma è stato scritto per QuickBASIC 4.x, ma chi lavora con BASIC PDS 7.x o VBDOS può seguire le direttive nelle note nel listato (vedi EvalExpression) e spostare il gestore degli errori dal modulo principale all'interno della procedura.

Il vantaggio di mantenere separata la fase di compilazione dalla fase di valutazione diventa evidente quando la stessa espressione deve essere ricalcolata per valori differenti di una variabile; per questo motivo il valutatore include due tavole interne varName$() e varValue#() che conservano nome e valore delle variabili usate nelle espressioni (da non confondere quindi con le variabili del programma BASIC), a cui i programmi applicativi possono accedere mediante la procedura DeclareVariable (per creare una nuova variabile) e SetVariable (per assegnare un valore ad una variabile dichiarata in precedenza). Ecco lo scheletro di un programma che stampa il valore di una espressione contenente la variabile "x", per i valori interi della variabile compresi tra 1 e 20

```
' dichiara la variabili "x" - in uscita xIndex% conterrà
' l'indice della variabile nelle tavole interne del valutatore
DeclareVariable xIndex%, "X"
' dopo aver dichiarato la variabile possiamo compilare
' l'espressione
compiled$ = CompileExpression$(expression$)
FOR x# = 0 TO 20
      ' prima di valutare l'espressione compilata dobbiamo
      ' assegnare il valore alla variabile "X", mantenendo il
      ' suo valore in sincrono con la variabile indice di questo
      ' ciclo FOR
      SetVariable xIndex%, x#
      result# = EvalExpression#(compiled$)
      PRINT "X =";x#;"   F(x)=";result#
NEXT
```

```
' al termine del programma è opportuno cancellare la variabile
' dalle tavole del valutatore, per non sprecare spazio
DeclareVariable xIndex%, ""
```

Naturalmente, è anche possibile definire più di una variabile, ad esempio

```
DeclareVariable xIndex%, "X"
DeclareVariable yIndex%, "Y"
```

I nomi delle variabili seguono le stesse regole del BASIC: il primo carattere deve essere una lettera, i caratteri rimanenti possono essere lettere o cifre, non viene fatta alcuna distinzione tra maiuscole e minuscole; poiché il vettore varName$() è composto da stringhe a lunghezza variabile non esiste un limite pratico alla lunghezza dei nomi; potete definire fino a cinquanta variabili, ma questo limite può essere facilmente modificato agendo sulla costante simbolica VARMAX nella sezione dichiarativa del modulo. Allo stesso modo, il valutatore accetta fino a 20 livelli di parentesi in sospeso, ma potete modificare tale limite (ad es. per risparmiare spazio in DGROUP) cambiando il valore della costante STACKMAX.

Il valutatore di espressioni supporta tutti gli operatori aritmetici del BASIC, e anche qualcuno in più

più e meno unari

qualsiasi numero di parentesi ()

gli operatori aritmetici ^ * / + - \ (divisione intera) MOD

gli operatori di confronto = <= < >= > <>

gli operatori booleani NOT AND OR XOR

le funzioni ad un argomento ABS INT FIX SGN SQR LOG EXP SIN COS TAN ATN

le funzioni a due argomenti MIN MAX

la funzione a tre argomenti IFF

Tra le funzioni "nuove" rispetto al BASIC standard ho incluso le funzioni MAX, MIN e IFF, sia perché esse sono molto utili, sia perché sono un buon esempio di funzioni a più argomenti; la loro sintassi è la seguente:

```
result = MIN(first, second)
result = MAX(first, second)
result = IFF(condition, first, second)
```

La funzione IFF restituisce il valore di **first** se la condizione restituisce un valore non nullo, altrimenti restituisce **second**; l'utilità di questa funzione è evidente, in quanto a differenza di quel che accade con i calcoli BASIC, la funzione da calcolare non può essere suddivisa in blocchi condizionali; ad esempio, la procedura BASIC

```
IF x# < 5 THEN
        result# = 1 / (x# - 5)
```

```
ELSE
        result# = -1
ENDIF
```

deve essere resa in questo modo con il valutatore di espressione

```
DeclareVariable xIndex, "x"
expr$ = "IFF(x < 5, 1/(x-5), -1)"
SetVariable xIndex, 1
result# = EvalExpression$(CompileExpression$(expr$))
```

Per testare sia la correttezza che la velocità del valutatore di espressioni, ho preparato un breve programma di demo che stampa una tavola di valori per qualunque funzione introdotta dall'operatore o scelta da una lista di espressioni predefinite; nel secondo caso il programma stampa anche i risultati della stessa espressione valutata usando il BASIC, e mostra un confronto dei tempi di elaborazione; nella maggior parte dei casi il valutatore eseguito sotto interprete è 2-3 volte più lento del BASIC, che non è affatto un cattivo risultato; le sorprese arrivano quando si compila il programma e si vede che il divario diminuisce sensibilmente, e in alcuni casi il valutatore è appena il 20% più lento del BASIC compilato.

```
'==================================================================
'              "EvalExpression" demo program
'==================================================================

'$INCLUDE: 'eval.bi'

DEFINT A-Z

DIM demo$(9)
DIM x AS DOUBLE, time1 AS SINGLE, time2 AS SINGLE

'Inizializza il vettore demo$()
FOR i = 1 TO UBOUND(demo$)
   READ demo$(i)
NEXT i

DO
   CLS
   PRINT "PROGRAMMA DEMO per 'EvalExpression()'"
   PRINT STRING$(64, "=")
   PRINT
   ' stampa le espressioni demo predefinite
   FOR i = 1 TO UBOUND(demo$)
      PRINT i, demo$(i)
   NEXT i
   PRINT
   PRINT "Scegli una delle espressioni  "
   PRINT "o introduci una nuova espressione - premi INVIO per terminare"
   PRINT
   LINE INPUT "F(x) = ", ex$
   IF ex$ = "" THEN EXIT DO

   n = 0
   IF LEN(ex$) = 1 THEN
      IF VAL(ex$) > 0 THEN
         n = VAL(ex$)
```

```
      ex$ = demo$(n)
   END IF
END IF

' X è la variabile nella espressione
x = 0
xIndex = 0
DeclareVariable xIndex, "X"
comp$ = CompileExpression$(ex$)
' ciclo di valutazione
' le medesime istruzioni funzionano con le espressioni predefinite
' e quelle introdotte dall'utente
DO
   CLS
   PRINT "TABLE OF VALUES :   F(x) = "; ex$
   IF n > 0 THEN
      PRINT TAB(10); "Eval#()"; TAB(40); "BASIC compilato"
   END IF
   PRINT STRING$(64, "=")

   FOR i = 1 TO 18
      PRINT "f("; LTRIM$(RTRIM$(STR$(x))); ") = ";
      ' Assegna il valore alla variabile e calcola l'espressione
      SetVariable xIndex, x
      value# = EvalExpression(comp$)
      PRINT TAB(10);
      IF EvalError THEN
         PRINT "Error # "; EvalError;
      ELSE
         PRINT value#; "    ";
      END IF
      ' se si tratta di una espressione predefinita, calcola la
      ' stessa espressione usando il BASIC
      IF n > 0 THEN
         ON n GOSUB sub1, sub2, sub3, sub4, sub5, sub6, sub7, sub8, sub9
         PRINT TAB(40); v#;
      END IF
      PRINT
      ' incrementa la variabile di controllo
      x = x + 1
   NEXT i


   ' stampa un'altra videata di dati se l'operatore lo richiede
   PRINT
   PRINT "Premi ESCAPE per terminare - un altro tasto per continuare";
   LOCATE CSRLIN, 1
   i$ = INPUT$(1)
LOOP WHILE i$ <> CHR$(27)

' se è una espressione predefinita, calcola il timing
IF n > 0 THEN
   PRINT "Aspetta alcuni secondi ...                                    ";
   iterations = 1000
   time1 = TIMER
   FOR i = 1 TO iterations
      v# = EvalExpression(comp$)
   NEXT i
   time1 = TIMER - time1
   time2 = TIMER
```

```
        FOR i = 1 TO iterations
          ON n GOSUB sub1, sub2, sub3, sub4, sub5, sub6, sub7, sub8, sub9
        NEXT i
        time2 = TIMER - time2
        IF time2 = 0 THEN time2 = 1 / 18.2
        LOCATE CSRLIN, 1
        PRINT " La funzione EVAL è";
        PRINT INT((time1 / time2) * 100) / 100; "volte più lenta del BASIC compilato"
        PRINT time1 / iterations; "secondi per iterazione -";
        PRINT " premi un tasto per continuare";
        PRINT SPACE$(79 - POS(0));
        i$ = INPUT$(1)
    END IF
LOOP

END

' subroutine per calcolare le espressioni predefinite
sub1: v# = 2 * 3 ^ x - x ^ 2 + 4: RETURN
sub2: v# = LOG(x ^ 2 + 10 * x + 3): RETURN
sub3: v# = COS(x) * x / (10 + 3.5): RETURN
sub4: v# = SQR(SIN(x) ^ 2 + COS(x) ^ 2): RETURN
sub5: v# = ATN(x ^ 2): RETURN
sub6: v# = x ^ INT(x / 4): RETURN
sub7: v# = x >= 5 AND x <= 10: RETURN
sub8: v# = x < 4 OR x ^ 2 > 200 OR x = 10: RETURN
sub9: v# = SIN(x) / (x ^ 2 + 1): RETURN

DATA "2 * 3 ^ x - x ^ 2 + 4"
DATA "LOG(x ^ 2 + 10 * x + 3)"
DATA "COS(x) * x / (10 + 3.5)"
DATA "SQR(SIN(x) ^ 2 + COS(x) ^ 2)"
DATA "ATN(x ^ 2)"
DATA "x ^ INT(x / 4)"
DATA "x >= 5 AND x <= 10"
DATA "x < 4 OR x^2 > 200 OR x = 10"
DATA "SIN(x) / (x ^ 2 + 1)"
```

COME FUNZIONA IL VALUTATORE DI ESPRESSIONI

Anche se non è necessario comprendere come funziona il valutatore di espressioni per poterlo usare efficacemente nei propri programmi, o persino per espanderlo con nuove funzioni (vedi più avanti), penso che i più curiosi potrebbero essere interessati all'algoritmo su cui esso si basa. Ho già mostrato come funziona un valutatore di espressioni RPN, e se confrontate la semplice procedura **EvalPRN** con la più completa **EvalExpression** noterete che esse sono in realtà molto simili: entrambe si basano su uno stack di valori numerici e ad ogni iterazione del ciclo principale eseguono una delle due operazioni seguenti:

- operazione di *push*, che pone un operando sullo stack (nome o variabile)
- operazione di *execute*, che preleva uno o più operandi dallo stack, applica

ad essi una particolare trasformazione matematica, ed infine memorizza il risultato nuovamente sullo stack

Non esiste una vera e proprio operazione di *pop*, poiché in questa "stack machine" possiamo accedere direttamente ai valori sullo stack, che altro non è che un vettore BASIC di numeri in doppia precisione. Per migliorare le prestazioni ho adottato un piccolo accorgimento: il valore in cima allo stack (Top-On-Stack) è memorizzato in una variabile semplice **Tos** e non è effettivamente spostato sullo stack fino a quando non occorre eseguire il push di un nuovo operando: in questo modo questo valore può essere letto e scritto senza l'overhead normalmente associato agli elementi di un array, ed inoltre si può accedere all'elemento successivo sullo stack (Next-On_Stack) usando l'espressione **Stak(sp)** anzichè **Stak(sp-1)** e risparmiando in tal modo alcuni cicli di CPU. Avendo discusso il processo di valutazione di una espressione RPN, possiamo concentrarci sul compito più difficile, e cioè la conversione di una espressione algebrica nel suo equivalente in notazione polacca inversa; ecco una breve spiegazione del funzionamento della routine **CompileExpression$**.

Come ho già fatto notare, il parser delle espressioni deve rimandare l'esecuzione delle operazioni matematiche fino a quando non sono stati letti tutti gli argomenti; nel nostro particolare caso non vogliamo realmente "eseguire" l'operazione - poiché stiamo compilando l'espressione e non valutandola - ma il processo è identico: l'unica differenza è data dal fatto che invece di eseguire le operazione dovremo aggiungere un codice (*opcode*) in coda alla stringa **compiled$**; al termine del parsing questa stringa conterrà gli operandi e gli operatori nell'ordine e nel formato atteso dal valutatore di espressioni RPN. Per operatori in sospeso il parser necessita di un'area dove memorizzare tutti gli opcode in sospeso: la migliore struttura per questo compito è un stack, che come al solito è simulato con un vettore e un puntatore al vettore stesso (lo stack pointer, variabile **opSp**). Ad esempio, l'espressione "12 + x" è convertita in quattro passi:

- analizza il primo operando, accoda in compiled$
- analizza il simbolo "+", convertilo in un opcode (costante simbolica opADD) e sistemalo sullo stack sullo stack degli operatori (operazione di push)
- analizza il nome della variabile "x", accoda in compiled$
- esegui il pop dell'opcode dallo stack, e accoda in compiled$

Il bisogno di usare uno stack anziché una semplice variabile per conservare gli operatori in sospeso diventa evidente quando l'espressione include più operatori; consideriamo la seguente espressione

```
12 + X * 4
```

in questo caso la moltiplicazione deve essere eseguita per prima anche se compare dopo l'addizione, a causa della sua priorità più alta; in questo caso il processo di parsing è più complicato

- parsing del primo operando "12", accoda in compiled$
- parsing del segno "+", push della costante opADD sullo stack degli operatori
- parsing del secondo operando "x", accoda in compiled$
- parsing del segno "*", push della costante opMUL sullo stack degli operatori
- parsing del terzo operando "4", accoda in compiled$
- pop dell'opcode della moltiplicazione, accoda in compiled$
- pop dell'opcode della addizione, accoda in compiled$

Una prima stesura dell'algoritmo potrebbe allora essere la seguente:

- quando viene incontrato un operando (numero o variabile) esso viene accodato a compiled$
- quando viene incontrato un operatore, la sua priorità è confrontata con la priorità dell'operatore in cima allo stack degli operatori; se risulta essere minore o uguale, l'operatore sullo stack viene accodato a compiled$ e scartato (pop) dallo stack stesso, e il confronto viene ripetuto con l'operatore che ora è in cima allo stack
- il passo precedente è ripetuto fino a quando vi sono operatori sullo stack e la loro priorità è maggiore o uguale alla priorità dell'operatore appena incontrato (detto anche l'operatore *corrente*), dopo di ché quest'ultimo viene posto in cima allo stack

L'algoritmo precedente funziona bene fino a quando non incontriamo una espressione con una o più coppie (eventualmente nidificate) di parentesi, come in

```
12 * (X + 4) ^ (X  / (4 +10))
```

Quando viene incontrata una parentesi aperta "(" il valutatore non può più usare nei confronto al passo (2) la priorità dell'operatore in cima allo stack, perché l'espressione tra parentesi deve avere comunque valutata per prima; tuttavia, è possibile preservare la semplicità dell'algoritmo aggiungendo solo alcuni punti:

- quando viene incontrata una parentesi aperta, il suo opcode è memorizzato immediatamente nello stack degli operatori, assegnando ad esso la massima priorità (255 nel nostro programma)

- quando si confronta l'operatore corrente con quello in cima allo stack, quest'ultimo deve essere scartato dallo stack e accodato in compiled$ se la sua priorità è maggiore o uguale a quella dell'operatore corrente AND l'operatore sullo stack non è una parentesi aperta
- quando si incontra una parentesi chiusa "(" le si assegna la priorità minima (zero) e si continua con il processo descritto al punto precedente: questo automaticamente farà in modo che tutti gli operatori nella sotto-espressione racchiusa tra parentesi saranno scartati dallo stack e accodati in compiled$;
- (al termine del processo indicato al punto precedente) se l'operatore in cima allo stack è una parentesi aperta AND l'operatore corrente è una parentesi chiusa, ambedue sono scartati senza accodare in compiled$, e si continua leggendo un nuovo operatore dalla espressione in forma algebrica

Questo è quanto: non è molto semplice, ma non è neanche difficile come molti di voi avranno immaginato; vi sono alcuni dettagli su cui sorvolo volentieri, visto che chi è realmente interessato può studiare il sorgente.

RICERCA DELLE RADICI DI UNA EQUAZIONE

Cosa si può fare con un valutatore di espressioni ? Come preannunciato nell'introduzione di questo capitolo, vi sono numerose applicazioni per una routine di questo tipo; forse una delle più semplici e scontate è il tracciamento di una funzione ad una o due variabili, ma si tratta di una applicazione quasi banale che lascio alla vostra iniziativa personale. Illustrerò invece un altro problema: la valutazione di tutte le radici di una equazione, un compito frequente in molti programmi scientifici.

Credo di poter dare per scontato che tutti voi sappiate risolvere semplici equazioni come le seguenti

```
4*X = 17                  (primo grado)
3*X^2 + 5*X - 27 = 0      (secondo grado)
X^4 + 2*X^2 - 12 = 0      (biquadratica, basta sostituire Y=X^2)
```

ma sono anche certo che molti entrerebbero in crisi anche solo con una semplice equazione di terzo grado come

```
X^3 + 7*X^2 - 4*X + 20 = 0
```

per non parlare di veri e propri "mostri" come

```
EXP(2 * SIN(X) + 1 / COS(X)) ^ 2 = 0
```

In realtà, a parte un piccolo sottoinsieme di tutte le possibili espressioni, la maggior parte delle equazioni possono essere risolte soltanto ricorrendo a

sistemi "numerici", che non sono altro che sofisticate tecniche "tenta-e-ritenta" in cui ad ogni iterazione ci si avvicina alla soluzione. Tra i tanti algoritmi sviluppati in questo campo ne ho scelto uno dei più semplici ed efficaci, detto "il metodo della secante"; ecco una breve descrizione dell'algoritmo:

- il metodo richiede una funzione F(x) continua, un intervallo di ricerca e la dimensione di un sotto-intervallo in cui, secondo noi, non vi può essere più di una radice; inoltre esso richiede una misura delle "tolleranza", da usare nella espressione -TOLERANCE <F(x)<+TOLERANCE in luogo del test F(x)=0, che potrebbe fallire a causa di errori di arrotondamento, ed un numero massimo di iterazioni.
- il programma comincia ad analizzare l'intervallo di ricerca, partendo dal suo limite inferiore e procedendo con un passo pari al sotto-intervallo definito in precedenza; quando il programma trova che la F(x) cambia segno il programma passa al punto (3), se invece la funzione non intercetta mai l'asse delle ascisse il programma termina con un messaggio appropriato
- ha trovato un sotto-intervallo, che chiameremo (X1,X2), in cui vi deve sicuramente essere una soluzione della equazione F(x)=0; per trovare una approssimazione di questa radice il programma calcola un terzo punto X3 compreso nel sotto-intervallo e che corrisponde alla intersezione della secante che unisce (X1,F(x1)) con (X2,F(X2))
- a seconda del segno di F(X3) si possono verificare tre sottocasi

a) ABS(F(X3)) < TOLERANCE : abbiamo trovato una radice

b) F(X3) ha lo stesso segno di F(X1); abbiamo trovato un sottointervallo più piccolo (X3,X2) che contiene la radice

c) F(X3) ha lo stesso segno di F(X2): abbiamo trovato un sottointervallo più piccolo (X1,X3) che contiene la radice

- il programma ripete i punti (3) e (4) fino a quando trova una radice oppure raggiunge il numero massimo di iterazioni consentite; in entrambi i casi X3 può essere considerata una ragionevole approssimazione della radice, e può essere quindi conservata in un vettore di soluzioni
- il programma ricomincia al punto (2), e continua fino a raggiungere l'estremo superiore dell'intervallo di ricerca

Notate che se la funzione F(x) è continua e definita per tutto l'intervallo di ricerca il programma può evitare di controllare errori come divisione per zero o argomenti non validi.

Basandomi su questo algoritomo ho scritto la procedura **SolveEquation**, che accetta in ingresso una espressione, i limiti inferiore e superiore dell'intervallo di ricerca e la dimensione del sottointervallo, e riempie un vettore con tutte le radici trovate

```
'$INCLUDE: 'eval.bi'

DEFINT A-Z
SUB SolveEquation (equation$, lowLimit#, highLimit#, interval#, rootsNum, roots#())
   '===================================================================
   ' Trova tutte le radici di una equazione in un dato intervallo
   '
   ' equation$    l'equazione che deve essere analizzata
   ' lowLimit#    limite inferiore dell'intervallo di ricerca
   ' highLimit#   limite superiore dell'intervallo di ricerca
   ' interval#    dimensione di un sotto-intervallo in cui supponiamo
   '              non vi possa essere più di una soluzione
   '
   ' rootsNum     il numero delle soluzioni trovate
   ' roots#()     un vettore che riporta tutte le soluzioni
   '===================================================================
   CONST MAXITERATIONS = 50
   CONST TOLERANCE# = 1D-30
   DIM compiled$, varIndex, counter, x1#, x2#, x3#, y1#, y2#, y3#

   ' crea la variabile "x" e compila l'espressione
   rootsNum = 0
   DeclareVariable varIndex, "X"
   compiled$ = CompileExpression$(equation$)
   IF EvalError THEN EXIT SUB
   x1# = lowLimit#
   SetVariable varIndex, x1#
   y1# = EvalExpression(compiled$)


   ' analizza l'intervallo di ricerca con passo pari a interval#
   DO
      x2# = x1# + interval#
      SetVariable varIndex, x2#
      y2# = EvalExpression(compiled$)

      IF SGN(y1#) <> SGN(y2#) THEN
         ' se troviamo un cambio di segno significa che abbiamo trovato
         ' un intervallo che contiene una radice dell'equazione
         ' possiamo calcolare una approssimazione di tale punto considerando
         ' l'intersezione della linea (x1,F(x1)) - (x2,F(x2)) con l'asse
         ' delle ascisse e ottenendo un terzo punto X3; se questo punto è
         ' ale che F(x3)=0 vuol dire che abbiamo trovato una radice,
         ' altrimenti possiamo considerarlo come il limite inferiore o
         ' superiore di un nuovo sottointervallo e ripetere l'operazione
         ' Per evitare dei loop infiniti causati dalla precisione finita
         ' dei calcoli floating point, abbiamo bisogno di porre un limite
         ' al numero di iterazioni

         FOR counter = 1 TO MAXITERATIONS
            x3# = x1# + y1# * (x2# - x1#) / (y1# - y2#)
            SetVariable varIndex, x3#
            y3# = EvalExpression(compiled$)
```

```
                ' a cuasa degli errori di arrotondamente dobbiamo evitare
                ' di testare Y3=0
                IF ABS(y3#) < TOLERANCE# THEN
                   EXIT FOR
                ELSEIF SGN(y1#) = SGN(y3#) THEN
                   x1# = x3#
                   y1# = y3#
                ELSE
                   x2# = x3#
                   y2# = y3#
                END IF
             NEXT

             ' uscendo da questo loop X3 contiene una "buona" approssimazione
             ' della radice dell'equazione
                 rootsNum = rootsNum + 1
             roots#(rootsNum) = x3#
          END IF

          ' continua ad analizzare l'intervallo di ricerca
          x1# = x2#
          y1# = y2#
          IF x1# > highLimit# THEN EXIT DO
       LOOP
    END SUB
```

Ecco un programma BASIC che sfrutta la routine

```
DIM roots#(20)
SolveEquation equation$, lowLimit#, highLimit#, interval#, rootsNum%, roots#()
IF rootsNum% = 0 THEN
      PRINT "Nessuna soluzione nell'intervallo indicato"
ELSE
      FOR i% = 1 TO rootsNum%
              PRINT "Solutione n.";i%;" is "; roots#(i%)
      NEXT

END IF
```

Notate che, per semplicità, in questa versione i valori di **TOLERANCE** e **MAXITERATIONS** sono stati introdotti nel codice come costanti simboliche, ma niente impedisce di passarli alla routine come argomenti per avere maggior controllo sulla velocità di esecuzione o sulla precisione dei risultati.

ESTENDERE IL VALUTATORE DI ESPRESSIONI

Se date una occhiata alla tavola degli opcode in testa al modulo EVAL.BAS vi potreste chiedere come mai le costanti numeriche sono definite in quel modo abbastanza inusuale: la ragione è che questa disposizione rende davvero semplice l'inserimento di nuovi codici in qualsiasi posizione della sequenza. Ad esempio, potete aggiungere i codici per ulteriori funzioni trigonometriche come ASIM o ACOS, o per funzioni "gestionali" come FV (Future Value), funzioni iperboliche (SINH, COSH, TANH, ecc.) e qualsiasi altro operatore che vi possa venir utile.

Aggiungere un nuovo opcode al valutatore di espressioni è un semplice processo in tre fasi:

- inserire il codice nella tavola degli opcode all'inizio del programma EVAL.BAS; il nuovo opcode dovrebbe essere inserito nella stessa sezione delle altre funzioni con il medesimo numero di argomenti; ad esempio, **opASIN** (funzione Arcoseno) potrebbe essere inserito immediatamente dopo **opTAN** nella sezione delle funzioni ad un argomento; notate che il programma tiene conto delle funzioni fino a quattro argomenti, anche se nessuna funzione del genere è correntemente definita; si raccomanda di usare una costante simbolica leggibile, e di attenersi alla convenzione "opXXXX"
- nella procedura **CompileExpression$** ricercate la sequenza "*****" ad aggiungete un altro blocco alla struttura CASE per riconoscere correttamente il nome della nuova funzione; usate le funzioni esistenti come linee guida
- nella procedura **EvalExpression** ricercate la sequenza "*****" ed aggiungete un blocco CASE che esegue la nuova funzione; notate che in questo punto dell'esecuzione della routine la variabile **Tos** contiene il valore dell'ultimo argomento della funzione, mentre **Stak(sp)** contiene il valore del penultimo argomento, **Stak(sp-1)** del terzultimo, e così via; in caso di dubbio potete "imitare" una funzione simile

Vi sono altre aree in cui il valutatore di espressioni potrebbe essere migliorato, specialmente dal punto di vista della velocità di esecuzione; per esempio, il valutatore di espressioni incluso in EXPERTO tratta anche con le variabili e le espressioni stringa, ed include inoltre una tecnica di ottimizzazione conosciuta come *constant folding*, che valuta durante la fase di compilazione tutte le sotto-espressioni costanti, come nel caso di "12*5/x" che può essere ovviamente trasformata in "60/x" risparmiando del tempo prezioso a runtime. Un'altra tecnica molto conosciuta è la cosiddetta *short-circuit evaluation*, anch'essa adottata da EXPERTO e dal compilatore BC.EXE fornito con il BASIC PDS e il Visual BASIC per Dos (ma non dall'ambiente di sviluppo QB/QBX/VBDOS o dal compilatore fornito con il QuickBASIC 4.x). La convenienza di questo tipo di ottimizzazione diventa evidente in una espressione booleana come:

```
X > 0 AND SQR(X * 10) = Y
```

dove, se X è minore o uguale a zero, il secondo operando di AND può essere completamente ignorato senza influenzare il risultato finale dell'espressione, che è ovviamente zero; inoltre, in questo modo si evita l'errore "Illegal function call" che deriverebbe dal tentativo di estrarre la radice quadrata di un numero negativo. Gli altri operatori che possono beneficiare da questa tecnica sono l'OR booleano, la moltiplicazione, la divisione e la funzione MOD.

Un altro modo di migliorare la performance del valutatore è di farlo lavorare in singola anziché in doppia precisione; ad esempio, in un programma che traccia il grafico di una funzione, la singola precisione è più che sufficiente. Per facilitare questa modifica ho scritto il programma in modo tale che sia sufficiente riscrivere alcune dichiarazioni indicate come AS DOUBLE, e naturalmente modificare il tipo di valore restituito da **EvalExpression**. Potete persino sostituire le routine aritmetiche del BASIC con delle vostre routine; ad esempio, la funzione di libreria del BASIC per l'elevamento a potenza non è molto efficiente quando l'esponente è un numero intero positivo, per cui potreste modificare il blocco corrispondente in **EvalExpression** con le righe seguenti:

```
CASE opPOWER
      IF Tos = 2 THEN
              Tos = Tos * Tos
      ELSEIF Tos = 3 THEN
              Tos = Tos * Tos * Tos
      ELSEIF Tos = 4 THEN
              Tos = Tos * Tos: Tos = Tos * Tos
      ELSE
              Tos = Stak(sp) ^ Tos
      END IF
      sp = sp - 1
```

Con tutti questi miglioramenti questo valutatore di espressioni è in grado di eguagliare la velocità del BASIC compilato, e in alcuni casi essere persino più veloce.

I PROGRAMMI DI RICERCA E L'ALGORITMO DI BACKTRACKING

Vi sono numerosi problemi, sia pratici che di natura puramente speculativa, che non si riescono a risolvere con algoritmi "numerici" al pari di quelli usati per risolvere una equazione o calcolare il determinante di una matrice. In questi casi l'unico metodo possibile è analizzare l'insieme delle possibili soluzioni per vedere quali risolvono il problema (problemi di enumerazione), e se necessario trovare quali soluzioni sono da considerarsi le migliori (problemi di ottimizzazione). Ecco alcuni esempi di problemi di questo tipo:

- trovare l'uscita da un labirinto
- determinare tutte le soluzioni del problema delle regine
- trovare il percorso più corto che unisce un insieme di città (il problema del commesso viaggiatore)
- giocare a dama o a scacchi, in cui ad ogni turno occorre trovare la "migliore" mossa tra quelle possibili
- determinare la più vantaggiosa combinazione di oggetti di dimensioni differenti che può essere contenuta in un recipiente (il problema dello zaino)
- ritagliare da un insieme di lastre (di metallo, di pelle, ecc.) un insieme di oggetti di forma differente, riducendo al minimo lo spreco di materiale

Pur nella loro eterogeneità, è evidente che in tutti i casi citati l'unico modo di risolvere il problema è di usare una strategia di tenta-e-ritenta, in modo da percorrere l'insieme di tutte le possibili soluzioni e trovare la prima che realmente soddisfa il problema, oppure memorizzare la migliore tra quelle incontrate.

Ad esempio, per risolvere il problema (a) l'unico sistema valido è visitare con metodo tutti i corridoi, girando sempre a destra quando arriviamo ad un incrocio, e marcando le strade percorse con un gessetto sempre sul lato destro del muro; quando arriviamo in un vicolo cieco dobbiamo ritenere di aver proseguito nella direzione sbagliata e torniamo indietro, continuando a tracciare una linea con il gessetto sul lato destro del muro. Quando arriviamo ad un incrocio già visitato prendiamo la prima strada mai imboccata; se tutte le strade risultano segnate con il gessetto dobbiamo ancora una volta riconoscere di aver sbagliato strada, e tornare lungo il corridoio segnato da un solo lato. In questo modo dovremmo trovare l'uscita, oppure ritornare al punto di partenza se il labirinto è senza uscite.

Nella descrizione precedente il punto interessante è diventa evidente che non è possibile o conveniente continuare in una direzione e decidiamo di tornare indietro fino all'incrocio precedente per provare un'altra strada. Questo modo di procedere e retrocedere, nel gergo informatico, si dice *backtracking*, e costituisce la base di tutti gli algoritmi di ricerca in grado di risolvere i problemi citati in precedenza, ed innumerevoli altri dello stesso tipo. Molte delle tecniche sviluppate nel settore dell'intelligenza artificiale si basano su questo concetto, come pure tutti i programmi che giocano a scacchi o a qualche gioco simile. La ricerca esustiva e il backtracking sono gli unici strumenti in grado di fornire una soluzione ottima al problema del commesso viaggiatore e al problema dello zaino, che peraltro sono esemplicazioni e astrazioni di problemi reali abbastanza comuni.

Prendiamo in esame il problema del commesso viaggiatore, che consiste nel trovare il percorso più corto (e quindi meno costoso) per viaggiare tra N città e tornare al punto di partenza. I percorsi possibili sono N * (N-1) * (N-2) * .... * 3 * 2 = N! (il fattoriale di N), che poi si riducono a (N-1)! se la città di partenza è stabilita. Purtroppo il fattoriale di un numero cresce ad una velocità vertiginosa: per una ricerca esaustiva su 10 città occorrere testare 362.880 percorsi differenti, e con appena venti città il totale dei percorsi sale ad oltre *due miliardi di miliardi*: per dare un'idea dell'enormità di questa cifra, un 80486 a 66 Mhz in grado di elaborare un percorso differente per ogni ciclo macchina impiegherebbe la bellezza di 1168 anni per trovare la soluzione al problema ! Questo è il lato che più di ogni altro mi affascina di questo tipo di problema: da una parte l'enormità delle quantità in gioco, dall'altra la necessità di spremere al massimo le "cellule grigie" (per dirla alla Hercule Poirot) per evitare di invecchiare davanti al computer.

IL PROBLEMA DELLE REGINE E DELLE AMAZZONI

Questo problema consiste nel sistemare otto regine sulla scacchiera in modo che non si mettano sotto scacco a vicenda. Sembra un giochino banale, ma pensate che il problema ha appassionato persino il grande Gauss e molti altri matematici meno famosi. La soluzione al problema non richiede un computer, ed infatti già nel secolo scorso qualcuno aveva avuto la pazienza di isolare le 92 soluzioni del problema. Ovviamente quello che è stato fatto manualmente può essere risolto più velocemente con l'ausilio del computer, e il "Problema delle regine" è rapidamente diventato un classico algoritmo di ricerca esaustiva e backtracking con cui si devono cimentare gli studenti al primo anno di Scienze dell'Informazione. Ecco come risolvere il problema con un algoritmo di ricerca esaustiva senza backtracking:

- genera tutte le possibili combinazioni di otto regine sulla scacchiera
- per ogni combinazione, testa che nessuna regina sia sotto scacco

Facendo qualche calcolo, la generazione di cui al punto (1) è un compito immane: si tratta infatti di oltre mille miliardi di possibili combinazioni. Avendo l'accortezza di sistemare una sola regina per riga le combinazioni possibili diminuiscono alla cifra di 16.7 milioni, che però sono tanti anche per un computer molto veloce. Usando il criterio di backtracking l'algoritmo è più complicato, ma può comunque essere riassunto in pochi punti:

- iniziare sistemando una regina sulla casella più a sinistra della prima riga
- sistemare una regina sulla seconda riga, nella prima casella non sotto scacco
- procedere in questo modo sistemando una regina sulla terza, quarta e quinta riga
- arrivati alla sesta riga (vedi figura 1) ci si rende conto che tutte le caselle sono occupate; evidentemente abbiamo imboccato una strada che non porta ad alcuna soluzione e occorre "tornare indietro"; questa è la fase del backtracking
- dobbiamo supporre allora che la regina in quinta riga debba essere mossa in qualche altra casella, ma essendo questa regina già sul bordo della scacchiera non abbiamo altre caselle a disposizione: siamo di nuovo in fase di backtracking
- se non vi è alcuna possibilità di sistemare una regina in quinta riga evidentemente dobbiamo provare un'altra casella per la regina in quarta riga; dalla figura si vede che la prima casella libera è nella settima colonna

FIGURA 13.1

R							
		R					
				R			
	R						
							R
?							

- a questo punto la fase di backtracking è (temporaneamente) terminata, e possiamo riprendere il nostro avvicinamento alla soluzione, sistemando la regina in quinta riga nella prima casella non sotto scacco (seconda colonna)
- continuiamo in questo modo avanzando di una riga quando riusciamo a trovare almeno una casella libera sulla riga stessa; se nessuna casella è libera retrocediamo alla riga precedente e spostiamo la regina di una casella a destra; se la regina si trova già sul bordo destro della scacchiera retrocediamo ancora di una riga, e così via
- quando riusciamo a sistemare una regina sulla ottava riga abbiamo trovato una soluzione e possiamo memorizzarla; la seguente figura mostra la prima soluzione del problema delle otto regine
- per trovare tutte le soluzioni successive attiviamo nuovamente il backtracking spostando la regina della penultima riga di una casella verso destra, e continuando con le solite regole
- il programma termina quando in fase di backtracking tentiamo di spostare la regina in prima riga oltre il bordo destro della scacchiera; la figura seguente indica l'ultima (la 92-esima) soluzione del problema

FIGURA 13.2

R							
				R			
							R
					R		
		R					
						R	
	R						
			R				

Il seguente programma traduce in BASIC l'algoritmo appena visto; la routine CalcolaRegine che esegue materialmente la ricerca è lunga meno di quaranta righe, commenti esclusi. Le altre routine servono per disegnare la scacchiera e mostrare il risultato sullo schermo.

FIGURA 13.3

							R
			R				
R							
		R					
					R		
	R						
						R	
				R			

```
DEFINT A-Z
DECLARE SUB DisegnaScacchiera ()
DECLARE SUB MostraScacchiera ()
DECLARE SUB CalcolaRegine ()


'_____________________________________
' dichiara le variabili globali in modo da renderle accessibili
' a tutte le procedure del programma senza doverle passare come argomenti
'_____________________________________
CONST MAXORDINE = 20
DIM SHARED rig(MAXORDINE)            ' usando un numero costante questi
DIM SHARED col(MAXORDINE)            ' array sono conservati in DGROUP e
DIM SHARED pri(MAXORDINE * 2)        ' velocizzano i calcoli
DIM SHARED sec(MAXORDINE * 2)
DIM SHARED max(MAXORDINE)
DIM SHARED oldCol(MAXORDINE)
DIM SHARED ordine                    ' ordine della matrice
DIM SHARED continua                  ' -1 se vi sono ancora soluzioni
DIM SHARED riga, colonna             ' coordinate di schermo
DIM SHARED soluzioni AS LONG         ' il numero di soluzioni trovate finora
DIM SHARED iterazioni AS LONG


ordine = 8
DisegnaScacchiera

'_____________________________________
' ciclo principale del programma
'_____________________________________
DO
   CalcolaRegine
   IF continua = 0 THEN EXIT DO
   soluzioni = soluzioni + 1
   LOCATE 24, 20: PRINT soluzioni;
   MostraScacchiera
LOOP
LOCATE 24, 40
PRINT "Iterazioni avvenute "; iterazioni
END

SUB CalcolaRegine STATIC
   '==================================================================
   ' Procedura principale di calcolo
   '
   ' Ad ogni nuova soluzione, la procedura imposta un valore non nullo
   ' nella variabile CONTINUA e torna al programma principale
   '==================================================================

   IF continua THEN
      ' rimuovi la regina in ultima riga e riprendi la ricerca di
      ' caselle libere alla sua destra
      ri = lato
      co = col(ri)
      ma = max(ri)
      rig(co) = 0
      pri(lato + ri - co) = 0
      sec(ri + co) = 0
      GOTO RegineNuovaColonna
   END IF
```

```
    ' imposta la variabile CONTINUA per le prossime chiamate
    continua = -1
    lato = ordine
    ' parti dalla prima riga (RI sarà incrementato)
    ri = 0

RegineNuovaRiga:
    ' passiamo ad analizzare la riga seguente
    ri = ri + 1
    ' la ricerca parte dalla colonna più a sinistra e termina
    ' al bordo destro della scacchiera
    co = 0
    ma = lato

RegineNuovaColonna:
    iterazioni = iterazioni + 1
    ' se siamo arrivati all'ultima colonna utile per questa riga
    ' (variabile MA) dobbiamo eseguire il backtracking
    IF co >= ma THEN
        ' fase di backtracking - torna alla riga precedente
        ri = ri - 1
        ' se tentiamo di retrocedere dalla riga zero non vi sono
        ' altre soluzioni e possiamo terminare
        IF ri = 0 THEN continua = 0: EXIT SUB
        ' i valori di MA e CO erano stati salvati in altrettanti vettori
        co = col(ri)
        ma = max(ri)
        rig(co) = 0
        pri(lato + ri - co) = 0
        sec(ri + co) = 0
        GOTO RegineNuovaColonna
    END IF

    ' sposta la regina sulla casella seguente, e continua
    ' fino a quando non si trova una casella la cui colonna e le
    ' due diagonali sono libere
    co = co + 1
    IF rig(co) THEN GOTO RegineNuovaColonna
    se = ri + co
    IF sec(se) THEN GOTO RegineNuovaColonna
    pr = lato + ri - co
    IF pri(pr) THEN GOTO RegineNuovaColonna

    ' se arriviamo a questo punto, la casella in riga RI-esima risulta
    ' libera (non sotto scacco) e possiamo "piazzare" la regina,
    ' segnalando che la sua colonna e le sue diagonali sono impegnate
    col(ri) = co
    rig(co) = ri
    pri(pr) = -1
    sec(se) = -1
    max(ri) = ma

    ' possiamo passare ad analizzare la riga successiva, distinguendo
    ' però il caso della regina nell'ultima riga della scacchiera
    IF ri < lato THEN GOTO RegineNuovaRiga

    ' torna al programma principale per segnalare una nuova soluzione
    EXIT SUB
END SUB
```

```
SUB DisegnaScacchiera
   '==================================================================
   ' Disegna la griglia vuota
   '==================================================================
   colonna = 39 - ordine
   riga = 3 - (ordine < 8) - (ordine < 11) - (ordine < 14) - (ordine < 17)
   CLS
   LOCATE riga, colonna
   PRINT " +"; STRING$(ordine * 2, 196); "+";
   FOR r = 1 TO ordine
      LOCATE riga + r, colonna
      PRINT " ¦"; SPACE$(ordine * 2); "¦";
   NEXT
   LOCATE riga + ordine + 1, colonna
   PRINT " +"; STRING$(ordine * 2, 196); "+";
   LOCATE 24, 10
   PRINT "soluzione: ";
   FOR i = 1 TO UBOUND(oldCol)
      oldCol(i) = 1
   NEXT
END SUB

SUB MostraScacchiera STATIC
   '==================================================================
   ' Mostra l'ultima soluzione trovata
   '==================================================================
   FOR r = 1 TO ordine
      LOCATE riga + r, colonna + oldCol(r) * 2
      PRINT "  ";
      LOCATE riga + r, colonna + col(r) * 2
      PRINT "__";
      oldCol(r) = col(r)
   NEXT
END SUB
```

Il vantaggio del backtracking rispetto alla ricerca esaustiva "pura" è di evitare la generazione di disposizioni che non possono portare a nessuna soluzione; proprio per mostrare il vantaggio della tecnica il programma precedente stampa il numero di volte che è stato necessario testare una casella per verificare se era libera o meno: si tratta di 17.685 iterazioni, contro le sedici milioni di combinazioni che avremmo dovuto testare se non avessimo introdotto la tecnica del backtracking.

Questo algoritmo iniziale può essere migliorato in più di un modo. Ad esempio, non è necessario spostare la prima regina oltre la quarta colonna, poiché tutte le soluzioni che troveremmo da quel momento in poi sarebbero riflessioni di una soluzione già trovata in precedenza; è possibile allore dimezzare il tempo di elaborazione fermando la ricerca in quel punto e raddoppiando il numero delle soluzioni trovate fino a quell'istante. Questa considerazione ci porta alla definizione di soluzioni base del problema delle regine, ossia tutte e sole le soluzioni realmente distinte, in cui cioè nessuna può essere ricavata dalle altre per mezzo di rotazioni o di riflessioni rispetto agli assi mediani o alle diagonali della scacchiera. Questa situazione però

non può considerarsi una soluzione base, in quanto risulta essere una rotazione di 90° in senso antiorario della soluzione mostrata in figura 2. Un algoritmo in grado di isolare solo le soluzioni base è decisamente più complesso di quello visto in precedenza, per cui eviterò di riportarlo nel testo: il sorgente commentato è nel file REGINE2.BAS; questo programma può essere inoltre generalizzato a qualsiasi ordine della scacchiera, anche se fareste bene a non tentare con ordini maggiori di 14 a meno che non disponiate di un 80486 o un Pentium ad almeno 50 Mhz e di tanto tempo libero.

Con l'andar del tempo qualcuno ha pensato di complicare ulteriormente il problema delle regine introducendo la variante delle amazzoni, che sono delle super-regine in grado di muoversi anche con il passo del cavallo. Il programma REGINE2.BAS è anche di calcolare le soluzioni per questo tipo di problema. In realtà il problema delle amazzoni si risolve più velocemente che il problema delle regine di pari ordine, poiché a causa della maggiore capacità offensiva delle amazzoni è più difficile trovare caselle libere e si deve ricorrere più frequentemente al backtracking. Il programma REGINE2.BAS è anche in grado di testare se una soluzione è base o meno prima di completare la disposizione di tutte le regine: il guadagno in velocità è di circa un ordine di grandezza, e il programma è in grado di affrontare in tempi ragionevoli il calcolo dell'ordine 16 per il problema delle regine, e dell'ordine 18 per le amazzoni. Il codice sfrutta tutti gli accorgimenti che sono riuscito a immaginare, e questo è davvero il massimo che sono riuscito ad ottenere dal BASIC; per spremere il massimo dalla CPU ho riscritto la routine CalcolaRegine in Assembly, e sono riuscito a risolvere il problema delle amazzoni di ordine 20 in "appena" 42 ore; se la notizia vi può interessare, esistono esattamente 61.984.976 modi distinti (soluzioni base) per sistemare venti amazzoni su una scacchiera senza che si minaccino a vicenda. I più curiosi e sfaccendati potranno controllare questo risultato con il programma REGINE3.EXE (fornito solo in forma eseguibile).

LO SVILUPPO DI SISTEMI RIDOTTI PER IL TOTOCALCIO

Chi di voi non ha mai desiderato scrivere un programma per lo sviluppo di sistemi ridotti per il Totocalcio, il Totip o l'Enalotto? A ben vedere, anche la generazione di un sistema ridotto può considerarsi un problema di ricerca nel campo delle soluzioni possibili per isolare quelle che rispondono ai nostri criteri - o correzioni, per usare il gergo degli addetti ai lavori. Nell'ambito del Totocalcio (o Totip, Enalotto, ecc.) per sistema integrale si intende l'insieme di tutte le colonne generate dal pronostico, che può contenere segni fissi, varianti doppie e varianti triple. In generale sono pochi a giocare realmente i sistemi integrali, che hanno la poco gradita caratteristica di costare parecchio;

ad esempio un pronostico con quattro doppie e quattro triple (e quindi cinque fisse) genera un sistema integrale di ben 1296 colonne.

I sistemi ridotti sono nati dalla considerazione seguente: supponiamo che il pronostico sia corretto e che la colonna vincente sia effettivamente prevista nel sistema integrale; la statistica insegna che però alcune colonne sono altamente improbabili, ad esempio quelle con più di cinque segni 2 o come meno di due segni 1; analogamente possiamo eliminare senza rimpianti tutte le colonne con più di cinque segni X consecutivi, e così via. Si dice allora che intendiamo ridurre il sistema integrale mediante una correzione sul numero minimo e massimo dei segni, e sul numero massimo di presenze consecutive dello stesso segno. A questo primo tipo di correzione se ne possono aggiungere innumerevoli altre, ma per adesso vediamo come sviluppare un programma per la riduzione di un sistema integrale.

```
'--------------------------------------
'    RIDUZIONE  SISTEMA  TOTOCALCIO
'--------------------------------------

DEFINT A-Z
DECLARE SUB SistemaRidotto ()
DECLARE SUB SistemaRidotto2 ()

CONST PARTITEMAX = 13

DIM SHARED pronostico$(PARTITEMAX)
DIM SHARED prono(PARTITEMAX, 4)
DIM SHARED segniMin(3), segniMax(3), segniMaxCons(3)

RESTORE
' il pronostico
DATA 12X, 12, 1X2, X12, X2, 12, X2, 12, 1X, 12X, X12, X2, X
' le presenze min, max e max consecutive per ciascun segno
DATA 3, 6, 13
DATA 0, 13, 13
DATA 2, 5, 4

' leggi il pronostico e le correzioni sui segni dalle istruzioni DATA
FOR i = 1 TO PARTITEMAX
   READ pronostico$(i)
NEXT
FOR i = 1 TO 3
   READ segniMin(i)
   READ segniMax(i)
   READ segniMaxCons(i)
NEXT

'--------------------------------------
'    Mostra il sistema sullo schermo
'--------------------------------------
CLS
PRINT "PRONOSTICI PARTITE"
PRINT
FOR i = 1 TO PARTITEMAX
   PRINT "Partita n."; i; TAB(20); pronostico$(i)
```

```
NEXT
PRINT
PRINT "PRESENZE SEGNI        minimo     massimo    massimo cons."
PRINT "——————————————————————————————————"
FOR i = 1 TO 3
   PRINT "              "; MID$("12X", i, 1); TAB(24);
   PRINT segniMin(i);
   PRINT TAB(35); segniMax(i);
   PRINT TAB(47); segniMaxCons(i)
NEXT

'——————————————————————————————————
'    Pre-elaborazione del sistema
'——————————————————————————————————
' calcola il numero di doppie e di triple, e delle colonne complessive
colonne& = 1
FOR i = 1 TO PARTITEMAX
   SELECT CASE LEN(pronostico$(i))
      CASE 1
      CASE 2
         doppie = doppie + 1
         colonne& = colonne& * 2
      CASE 3
         triple = triple + 1
         colonne& = colonne& * 3
      CASE ELSE
         PRINT "Errore !": END
   END SELECT
NEXT
LOCATE 2, 40
PRINT "Doppie:"; doppie; "   Triple:"; triple
LOCATE 4, 40
PRINT "Colonne sistema integrale:"; colonne&
LOCATE 6, 40
PRINT "Colonne sistema ridotto:  ";
' trasforma i pronostici in una matrice
FOR i = 1 TO PARTITEMAX
   FOR j = 1 TO LEN(pronostico$(i))
      prono(i, j) = INSTR("12X", MID$(pronostico$(i), j, 1))
   NEXT
NEXT
CALL SistemaRidotto
END

SUB SistemaRidotto
   '——————————————————————————————
   ' Calcola il numero di colonne del sistema ridotto
   ' lo scarto delle colonne avviene a colonna ultimata
   '——————————————————————————————
   DIM indice(PARTITEMAX), partita, colonne&
   DIM colonna(PARTITEMAX)
   DIM presenze(3), segniConsecutivi

PartitaSuccessiva:
   ' passa ad analizzare la partita successiva
   IF partita = PARTITEMAX THEN
      GOSUB ControllaColonna
      GOTO SimboloSuccessivo
   END IF
   partita = partita + 1
```

```
      indice(partita) = 1
      simbolo = prono(partita, indice(partita))
      GOTO AggiornaColonna

   SimboloSuccessivo:
      indice(partita) = indice(partita) + 1
      simbolo = prono(partita, indice(partita))
      IF simbolo = 0 THEN GOTO Backtracking

   AggiornaColonna:
      colonna(partita) = simbolo
      GOTO PartitaSuccessiva

   Backtracking:
      IF partita = 1 THEN EXIT SUB
      partita = partita - 1
      GOTO SimboloSuccessivo

   ControllaColonna:
      ' calcola le presenze di ciascun simbolo, e il numero di
      ' presenze consecutive
      presenze(1) = 0
      presenze(2) = 0
      presenze(3) = 0
      segniConsecutivi = 0

      FOR i = 1 TO PARTITEMAX
         simbolo = colonna(i)
         presenze(simbolo) = presenze(simbolo) + 1
         IF simbolo <> colonna(i - 1) THEN
            segniConsecutivi = 1
         ELSE
            ' scarta le colonne con troppi segni consecutivi
            segniConsecutivi = segniConsecutivi + 1
            IF segniConsecutivi > segniMaxCons(simbolo) THEN RETURN
         END IF
      NEXT
      ' scarta le colonne con troppi o troppi pochi simboli
      FOR simbolo = 1 TO 3
         IF presenze(simbolo) < segniMin(simbolo) THEN RETURN
         IF presenze(simbolo) > segniMax(simbolo) THEN RETURN
      NEXT

      ' la colonna è stata accettata !
      colonne& = colonne& + 1
      LOCATE 6, 64: PRINT colonne&
      ' in questo punto è possibile stampare la colonna trovata
      ' oppure memorizzarla in un array..
      RETURN
   END SUB
```

Come vedete, i dati del problema (pronostico e correzione) sono inseriti nel programma sotto forma di istruzioni DATA; in un programma commerciale essi sarebbero introdotti interattivamente dall'operatore, ma per ora il mio obiettivo era mostrare l'algoritmo di ricerca. Se studiate con attenzione i commenti nella procedura SistemaRidotto, vedrete che in realtà il programma genera le soluzioni del sistema integrale dalla prima alla tredicesima riga, e solo allora controlla se le correzioni richieste impongono di scartare o

accettare la colonna così generata. Di fatto, quindi, il programma usa un algoritmo di ricerca senza l'uso del backtracking. Il programma completo riportato sul dischetto contiene anche la routine SistemaRidotto2, che invece usa attivamente il backtracking e riesce a ridurre il tempo di elaborazione ad una frazione del tempo richiesto dalla routine non ottimizzata.

```
SUB SistemaRidotto2
    '______________________________
    ' Calcola il numero di colonne del sistema ridotto
    ' lo scarto delle colonne avviene mentre sono colcolate
    '______________________________
    DIM indice(PARTITEMAX), partita, colonne&
    DIM colonna(PARTITEMAX)
    DIM consecutivi(PARTITEMAX)
    DIM presenzeMin(PARTITEMAX, 3)
    DIM presenze(3)

    ' inizializza la matrice presenzeMin() con il numero minimo di
    ' presenze per ciascun simbolo e per ciascuna delle tredici righe
    presenzeMin(PARTITEMAX, 1) = segniMin(1)
    presenzeMin(PARTITEMAX, 2) = segniMin(2)
    presenzeMin(PARTITEMAX, 3) = segniMin(3)
    FOR i = PARTITEMAX - 1 TO 1 STEP -1
       ' copia le presenze minime dalla riga successiva, ma sottrai uno
       ' se il simbolo appare nella riga i-esima del pronostico
       presenzeMin(i, 1) = presenzeMin(i + 1, 1) + (INSTR(pronostico$(i), "1") > 0)
       presenzeMin(i, 2) = presenzeMin(i + 1, 2) + (INSTR(pronostico$(i), "2") > 0)
       presenzeMin(i, 3) = presenzeMin(i + 1, 3) + (INSTR(pronostico$(i), "X") > 0)
    NEXT

PartitaSuccessiva2:
    ' passa ad analizzare la partita successiva
    IF partita = PARTITEMAX THEN
       GOSUB AccettaColonna2
       GOTO RimuoviSimbolo2
    END IF

    partita = partita + 1
    indice(partita) = 1
    simbolo = prono(partita, indice(partita))
    GOTO AggiornaColonna2

RimuoviSimbolo2:
    simbolo = colonna(partita)
    presenze(simbolo) = presenze(simbolo) - 1

SimboloSuccessivo2:
    indice(partita) = indice(partita) + 1
    simbolo = prono(partita, indice(partita))
    IF simbolo = 0 THEN GOTO Backtracking2

AggiornaColonna2:
    ' controlla che non vi siano già troppi simboli
    IF presenze(simbolo) >= segniMax(simbolo) THEN
       GOTO SimboloSuccessivo2
    END IF
```

```
   ' controlla che non vi siano troppi simboli consevcutivi
   IF simbolo <> colonna(partita - 1) THEN
       consecutivi(partita) = 1
   ELSE
      ' se il simbolo è uguale a quello nel rigo precedente
      ' controlla di non superare il massimo consentito
      IF consecutivi(partita - 1) >= segniMaxCons(simbolo) THEN
         GOTO SimboloSuccessivo2
      END IF
      consecutivi(partita) = consecutivi(partita - 1) + 1
   END IF
   ' infine controlla che non vi siano troppi pochi simboli
   IF simbolo <> 1 THEN
       IF presenze(1) < presenzeMin(partita, 1) THEN GOTO SimboloSuccessivo2
   END IF
   IF simbolo <> 2 THEN
       IF presenze(2) < presenzeMin(partita, 2) THEN GOTO SimboloSuccessivo2
   END IF
   IF simbolo <> 3 THEN
       IF presenze(3) < presenzeMin(partita, 3) THEN GOTO SimboloSuccessivo2
   END IF
   ' aggiorna il conto delle presenze per questo simbolo
   colonna(partita) = simbolo
   presenze(simbolo) = presenze(simbolo) + 1
   GOTO PartitaSuccessiva2

Backtracking2:
   IF partita = 1 THEN EXIT SUB
   partita = partita - 1
   GOTO RimuoviSimbolo2

AccettaColonna2:
   ' la colonna è stata accettata !
   colonne& = colonne& + 1
   LOCATE 11, 64: PRINT colonne&
   ' in questo punto è possibile stampare la colonna trovata
   ' oppure memorizzarla in un array..
   RETURN
END SUB
```

La routine SistemaRidotto2 controlla che le condizioni imposte dalle correzioni siano verificate durante la generazione di ciascuna colonna, ed è quindi in grado di attivare il backtracking qualunque sia la riga in esame; ovviamente, se una condizione risulta violata nelle prime fasi della costruzione della colonna (ossia per un valore basso della variabile partita) il backtracking sarà in grado di scartare un maggior numero di colonne del sistema integrale senza doverle analizzare una per una. Delle tre correzioni statistiche, la più facilmente verificabile è quella che impone un limite superiore alle presenze di ciascun segno; è quindi sufficiente testare che questo limite non sia mai superato:

```
IF presenze(simbolo) >= segniMax(simbolo) THEN ...
```

più difficile è il controllo sul numero dei segni consecutivi; per eseguire tale controllo con la massima efficienza il programma costruisce e mantiene aggiornato il vettore **consecutivi()**, che per ciascuno dei livelli già visitati

conserva il numero di segni consecutivi fino a quel livello, oppure 1 se il simbolo al livello precedente è differente da quello al livello attuale:

```
IF simbolo <> colonna(partita - 1) THEN
     consecutivi(partita) = 1
ELSE
   IF consecutivi(partita - 1) >= segniMaxCons(simbolo) THEN
      GOTO SimboloSuccessivo2
   END IF
   consecutivi(partita) = consecutivi(partita - 1) + 1
END IF
```

Infine, per controllare che in ogni momento sia verificato il numero minimo di presenze per ciascun segno, occorre quindi precalcolare una matrice **presenzeMin(PARTITEMAX, 3)** che per ciascun livello indica qual è il numero minimo di presenze per ciascun segno; i commenti nel listato dovrebbero essere sufficienti a capire come funziona l'algoritmo.

Il programma TOTORID.BAS riportato sul dischetto è lungi dall'essere un programma completo; manca una buona interfaccia utente, la visualizzazione delle colonne e la stampa sulle schedine, ma soprattutto mancano molte delle riduzioni che contraddistingono un programma professionale. Ecco alcune correzioni che potreste aggiungere di vostro pugno.

LA CORREZIONE DEGLI ERRORI E DELLE SORPRESE

Un *errore* è un segno che appare come seconda scelta in una variante doppia, una *sorpresa* è il segno che appare come terza scelta in una tripla; correggere un sistema imponendo ad esempio 2-4 errori e 1-2 sorprese significa eliminare tutte le colonne del sistema integrale che comprendono un numero minore o maggiore degli errori e delle sorprese previste.

LA CORREZIONE PER FASCE DI PROBABILITÀ

Ad ogni segno in ciascuna partita è possibile associare una probabilità (ovviamente stimata dal creatore del sistema), ad esempio:

E' allora possibile filtrare le colonne in base alla probabilità composta dei segni contenuti in essa, ad esempio decidendo di scartare le colonne con probabilità minore di 0.001% e maggiore di 0.01%; tenete a mente che la probabilità complessiva è calcolata moltiplicando tra loro le probabilità dei singolo segni.

LA CORREZIONE SUL NUMERO DI INTERRUZIONI

Questa correzione è in un certo senso l'opposto della correzione sul numero di segni consecutivi, e scarta tutte le colonne con più di un numero prefissato di *interruzioni*, ossia di cambiamenti di segno da una riga all'altra; ad esempio, la colonna 11XX21XX1112X contiene sette interruzioni.

Ogni sistemista può inventare le correzioni che più gli piacciono, anche se in alcuni casi esse potrebbero non avere alcuna rilevanza statistica; difatti, i programmi professionali per la riduzione dei sistemi sono diventati estremamente complicati e difficili da usare per chi non è un "addetto ai lavori". TOTORID è un programma semplice ed efficace, e può essere modificato secondo i propri gusti e preferenze; se per puro caso riuscite a fare tredici ricordatevi di mandarmi almeno una cartolina...

APPENDICE

Il mio principale lavoro consiste nel produrre programmi per altri programmatori; per poterlo svolgere nel migliore dei modi qualche anno fa ho creato la SoftWhale, che produce e distribuisce in tutta Italia librerie e utility per facilitare il lavoro degli sviluppatori, e offre servizi di consulenza a ditte e software house; siamo specializzati nella produzione di routine in BASIC e in Assembly super-ottimizzate, più veloci e compatte di qualsiasi procedura scritta in un linguaggio ad alto livello. Attualmente il nostro catalogo include quattro prodotti.

QBKIT

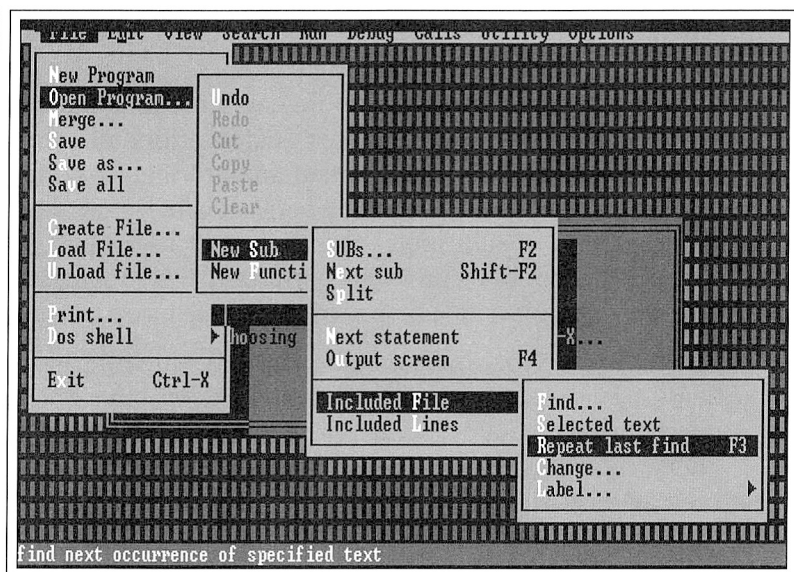
QBKIT è una libreria per Microsoft BASIC che comprende oltre 750 (settecentocinquanta!) nuovi comandi. La maggior parte di queste routine sono scritte in Assembly, ma non avrete bisogno di conoscere questo linguaggio per poterle sfruttare nei programmi: sarà sufficiente chiamare una procedura o una funzione. E' noto che l'Assembly è il linguaggio più veloce e compatto, e la combinazione BASIC+Assembly produce uno dei più potenti strumenti per programmare in ambiente MsDos. **QBKIT** permette di creare programmi che non potrebbero mai essere scritti in BASIC "puro". Potrete aprire una finestra che raccoglie l'output dei programmi esterni e dei comandi Dos; salvare e ricaricare il contenuto del video in qualsiasi modo grafico;

conoscere tutti i dati di configurazione del sistema (tipo di CPU e coprocessore, scheda video, porte seriali e parallele, versione Dos, software di rete, memoria cmos, ecc.). Oltre alle normali operazioni sui file, **QBKIT** permette di caricare e salvare interi vettori e matrici con un'unica operazione, di leggere interi alberi di sottodirectory, e persino di modificare direttamente i settori del disco. Inoltre, poiché le routine non richiedono l'istruzione ON ERROR, i programmi eseguibili sono più compatti e più veloci: su un sistema 386/25 **QBKIT** esegue l'ordinamento di un vettore di 10000 elementi in meno di un secondo e crittografa un megabyte di dati in due-tre secondi. Per creare programmi moderni e funzionali, **QBKIT** include un sistema completo di finestre tridimensionali; in ciascuna finestra si può creare una dialog box con campi testo, checkbox, listbox (anche con selezioni multiple) option button e bottoni di comando. Si può disporre di un sistema completo di menù a tendina in stile Lotus 123, menù popup che appaiono in qualunque posizione dello schermo, e menù pulldown, con il supporto del mouse, hotkey, shortcut key, messaggi sulla riga inferiore dello schermo, elementi non selezionabili, ecc. Le routine di interfaccia utente e le altre routine in BASIC sono fornite in sorgente (circa 300K); i sorgenti delle routine Assembly (circa 20mila righe di codice) possono essere acquistati separatamente.

Ai più esigenti offriamo una versione potenziata della libreria che permette di trattare con megabyte di dati numerici e di stringhe e di sfruttare fino all'ultimo byte della memoria installata sul sistema e di aggirare una delle più noiose limitazioni del BASIC sulle stringhe - che è noto non possono occupare complessivamente più di circa 45K (in QuickBASIC) o 192K (con BASIC PDS e Visual BASIC per Dos). Con **QBKIT PRO** potrete creare vettori di stringhe che occupano tutta la memoria convenzionale, e persino sfruttare tutta la memoria superiore (UMB), espansa (EMS) ed estesa (XMS) per conservare i vostri vettori e qualsiasi altro tipo di dati. **QBKIT PRO** è l'<u>unica</u> Quick Library che può essere caricata nei blocchi UMB, in modo da lasciare più spazio a disposizione del programma e dei dati. Per guadagnare tempo durante la compilazione del programma è possibile usare una funzione di MAKE integrata nell'ambiente di sviluppo, che ricompila solo i file sorgenti modificati dopo l'ultima compilazione. Infine, per i progetti più complessi potete utilizzare un <u>preprocessore</u> per i programmi BASIC, che permette ad esempio di mantenere un solo sorgente per più versioni dello stesso programma.

DSWAPPER

Una utility per a tutti i programmatori che si scontrano abitualmente contro il limite dei 640K imposti dal sistema operativo. Un aspetto particolare di

Menu pulldown multilivello

questo problema è l'impossibilità pratica di lanciare un programma dall'interno di un'altra applicazione, non essendovi spesso sufficiente memoria per entrambi; **dSWAPPER** è stato progettato per risolvere questo tipo di problema, rendendo possibile l'esecuzione di un programma qualsiasi dall'interno di una applicazione mediante lo *swapping* del codice e dei dati del programma principale su memoria espansa o su disco. Ecco in sintesi le principali caratteristiche di dSWAPPER:

A differenza di altre librerie per programmatori in commercio, dSWAPPER funziona con qualsiasi linguaggio di programmazione, e soprattutto non richiede modifiche al programma sorgente né ricompilazioni; questa utility funziona anche con i programmi di cui non si dispone il sorgente, incluso quindi word processor, tabelloni elettronici, database, programmi di grafica, ecc. dSWAPPER è scritto in Assembly, in modo da essere estremamente compatto e veloce; questo significa che le prestazioni del programma principale non saranno degradate in alcun modo. Il programma permette di forzare un programma ad allocare memoria oltre il limite dei 640K (l'area detta memoria riservata oppure blocchi UMB); tra i programmi si avvantaggiano di questa caratteristica vi è proprio l'ambiente di sviluppo del Microsoft BASIC. dSWAPPER richiede solo 10K di memoria convenzionale, e può essere caricato in alta memoria mediante l'istruzione LOADHIGH per non sottrarre neanche un byte al programma applicativo. E' possibile "inviare" un numero qualsiasi di tasti al programma esterno, in modo da eseguire qualunque operazione senza l'intervento dell'utente; è anche possibile impostare delle pause tra i tasti. Il pacchetto include una versione runtime che può essere liberamente distribuita con i propri programmi applicativi.

BATCH WIZARD

I programmi batch sono da sempre uno strumento indispensabile per programmatori ed utenti esperti, per l'automazione del lancio di applicativi, backup degli archivi, menù e programmi di installazione; è però vero che l'utente è diventato più esigente, e per soddisfare le sue aspettative i file batch standard sono insufficienti, lenti e con un look ben poco professionale. L'alternativa è di usare un linguaggio ad alto livello come il C o il Pascal, la cui potenza e complessità è però sproporzionata a compiti di questo tipo.

BATCH WIZARD riporta la programmazione batch al passo coi tempi, estendendo il Dos con numerosi nuovi comandi e con istruzioni tipiche di linguaggi più evoluti senza però perdere la semplicità e la facilità di apprendimento del linguaggio batch standard. Il compilatore tratta senza problemi anche i programmi scritti per DR DOS e per i batch enhancer 4DOS e Norton NDOS, e compila correttamente costrutti che mettono in crisi tutti gli altri compilatori batch in commercio, ad es. le istruzioni GOTO %label% oppure CALL %1. E' stato superato il limite dei 127 caratteri per riga, si possono nidificare le istruzioni FOR, usare i costrutti IF..ELSEIF e DO..LOOP, creare subroutine, variabili locali, procedure ricorsive, concatenare più istruzioni sulla stessa riga. Sono pienamente supportate le espressioni aritmetiche e stringa, con un centinaio di funzioni per accedere alla configurazione di sistema (versione Dos, cpu e coprocessore, spazio su disco, scheda video, memoria ram/ems/xms, ...).

Con BATCH WIZARD basta una singola istruzione per creare una finestra con cornice, ombra, effetto esplosione e salvataggio dello schermo sottostante; analogamente per numerare e stampare tutte le righe non vuote di un file di testo è sufficiente un programma di una sola riga! Per facilitare la fase di test il pacchetto include un interprete, che permette l'esecuzione passo-passo del programma, l'impostazione di breakpoint e la visualizzazione del valore delle variabili. Quando il programma è stato testato a dovere, il compilatore assembla tutti i programmi batch richiamati per mezzo di CALL e produce un eseguibile stand alone e liberamente distribuibile.

Un programma che cancella (o copia, comprime, crittografa, ecc.) **tutti i file selezionati dall'operatore richiede DUE istruzioni !**

La prima versione di BATCH WIZARD ha riscosso un notevole successo tra i nostri utenti, ed ha fruttato numerose segnalazioni e recensioni su riviste specializzate, tra cui PCMagazine e Computer Programming; la nuova versione 2.10 aggiunge funzioni di crittografia dati, cancellazione a prova di undelete,

browser per file di testo, menù popup e dialog box con hotkey, supporto del mouse ed help contestuale, un completo sistema di window in grado di "catturare" l'output di programmi esterni, screen saver e blocco di tastiera con password, un orologio digitale multitasking, e persino la modifica dei valori di FILES e LASTDRIVE senza editare config.sys e senza resettare il sistema. La versione PRO aggiunge inoltre: swapping su memoria espansa o su disco, per non sottrarre memoria agli applicativi lanciati dal programma batch; macro di tastiera lunghe fino a 32mila tasti, per automatizzare sequenze di operazioni all'interno di programmi esterni; compressione di file per operazioni di backup (anche su più dischetti) o per creare sofisticati programmi di installazione. La procedura di installazione dello stesso BATCH WIZARD è un file batch compilato, fornito anche in forma sorgente per poter essere studiato e modificato per le proprie esigenze. Il manuale di 250 pagine riporta numerosi esempi, e il nostro servizio di assistenza tecnica è sempre a vostra disposizione .

FIGURA 2

Una dialog box complessa, con campi, bottoni e look 3D

```
BWI.EXE          92436  02/05/93  02:00:00  ...A
BW.OVL           45454  02/05/93  02:00:00  ...A
SWAP.COM         11080  02/05/93  02:00:00  ...A
BWC.EXE          92402  02/05/93  02:00:00  ...A
MOVE.EXE         12438  02/05/93  02:00:00  ...A
SCRSAVER.EXE      7388  02/05/93  02:00:00  ...A
PATCHTSR.EXE     10324  02/05/93  02:00:00  ...A
STAMPA.EXE       11174  02/05/93  02:00:00  ...A
LEGGIMI.          3645  02/05/93  02:00:00  ...A
NOVITA.BW2       12038  02/05/93  02:00:00  ...A
BW0.LIB          69738  02/05/93  02:00:00  ...A
BW1.LIB          63524  02/05/93  02:00:00  ...A
BW2.LIB          59894  02/05/93  02:00:00  ...A
BW3.LIB          53612  02/05/93  02:00:00  ...A
BW4.LIB          50018  02/05/93  02:00:00  ...A
BW5.LIB          43720  02/05/93  02:00:00  ...A
BW6.LIB         105112  02/05/93  02:00:00  ...A
BW7.LIB          79624  02/05/93  02:00:00  ...A
TEMP.BAT            77  14/05/92  12:07:32  ...A
```

NOWAY

Produrre software costa tempo, fatica e denaro, ed è un desiderio naturale voler tutelare questi sforzi e evitare la perdita di profitti dovuta alla circolazione di copie illegali. La maggior parte dei metodi di protezione attualmente in commercio appartiene ad una delle due seguenti categorie: da una parte quelli basati su dischetti non copiabili usando i comandi standard del Dos, dall'altra quelli che si affidano ad una chiave hardware; entrambi i metodi hanno pregi e difetti. I dischetti sono relativamente economici, ma possono risultare non compatibili con alcuni disk drive, sono soggetti a deterioramento e comunque sono facilmente aggirabili da un "hacker" con un minimo di esperienza, tant'è vero che da anni nessuna software house d'oltreoceano usa più questo

sistema di protezione. Le chiavi hardware sono più sicure ma anche sensibilmente più costose, e anch'esse possono creare problemi di compatibilità con stampanti ed altre periferiche collegate alla porta parallela o seriale, specialmente quando si collegano in cascata chiavi di produttori differenti.

NOWAY è un nuovo sistema di protezione software che si basa su criteri differenti da quelli appena descritti, funziona con tutti i linguaggi di programmazione compilati, non richiede modifiche al sorgente né ricompilazioni. Il concetto alla base del sistema è semplice: per ogni computer è possibile determinare una *firma* che lo distingue da qualsiasi altro sistema, anche da identici modelli della stessa marca. Per funzionare regolarmente i programmi protetti devono essere "abilitati" per un particolare sistema; questa operazione può essere effettuata presso il cliente dal programmatore o dal personale della software house, oppure direttamente dall'utente finale, che comunica la firma di sistema e ne riceve in cambio una password. Questa operazione deve essere eseguita una volta soltanto, dopo aver copiato i file su disco rigido: ad ogni esecuzione successiva **NOWAY** controlla che firma e password corrispondano, e si rifiuta di eseguire il programma in caso contrario. Poiché si basa esclusivamente su caratteristiche documentate del Dos, **NOWAY** non è soggetto ai problemi di compatibilità che affliggono altri metodi di protezione; per default le caratteristiche che determinano la firma del sistema non risentono di operazioni quali la deframmentazione dei file e la compressione del disco mediante DoubleSpace, Stacker o simili, ma è' comunque possibile stabilire il grado di flessibilità con cui **NOWAY** si deve adattare ad eventuali variazioni della configurazione del sistema. Se si rende necessaria una reinstallazione del programma protetto l'autore del software può controllare a distanza - confrontando la firma originale con la nuova firma comunicata dall'utente - quali cambiamenti sono avvenuti nel sistema, e decidere se fornire una nuova password o meno.

Oltre a proteggere i programmi dalla copia, **NOWAY** è in grado di impedire il reverse engineering: poiché il file eseguibile è crittografato non è possibile modificare i messaggi di copyright né attuare la decompilazione del codice per risalire al sorgente (tecnica usata comunemente per sproteggere i programmi scritti in Clipper); poiché spesso è poco pratico codificare il nome dell'utente nel file sorgente **NOWAY** permette la crittografia di un file di dati che potrà essere letto soltanto dal programma protetto. In tal modo si possono proteggere efficacemente programmi scritti con linguaggi interpretati come il dBASE e le versioni di COBOL che non producono veri eseguibili. E' anche possibile imporre il controllo del CRC del file eseguibile. Per i programmi a grande diffusione, che non rendono praticabile la fornitura di una password via telefono a ciascun utente, **NOWAY** permette di stabilire la password durante la crittografia; è chiaro che questo meccanismo non è un sistema di protezione

vero e proprio, ma di fatto esso costituisce un efficace deterrente contro la proliferazione incontrollata delle copie illegali. Questo è il metodo usato attualmente da produttori come Microsoft, Quaterdeck, ecc. **NOWAY** permette anche di creare programmi dimostrativi in grado di funzionare per un determinato numero di minuti, oppure di impostare un numero massimo di esecuzioni per ciascuna applicazione, in modo da tutelarsi da possibili insolvenze da parte di un cliente; in entrambi i casi questi programmi demo possono essere convertiti in versioni definitive semplicemente fornendo una nuova password. Poiché funziona con qualsiasi file eseguibile, **NOWAY** può anche essere usato dal responsabile di un sistema informativo per accertarsi che i programmi acquistati non siano portati all'esterno dell'azienda.

NOWAY è in grado di proteggere programmi EXE per MsDos, e non funziona con i file COM; non esiste al momento una versione per Ms-Windows. Il processo di protezione è molto semplice e richiede pochi secondi. I programmi protetti possono funzionare in rete locale e su computer remoto, a condizione che la abilitazione sia effettuata per ciascuna stazione della rete, e che ogni sistema disponga di un disco rigido locale. E' possibile abilitare più programmi con la medesima password oppure abilitare singolarmente differenti funzioni di un applicativo, purché a ciascuna funzione corrisponda un file eseguibile distinto. Per quanto ne so è l'unico programma del suo genere che non richiede un pagamento aggiuntivo per ciascuna protezione effettuata.

IL SOFTWARE SU DISCHETTO

Il dischetto che accompagna questo testo contiene i file sorgenti di tutte le routine presentate nei vari capitoli, già testate e pronte all'uso; in generale tutte le routine di un medesimo capitolo sono raggruppate in un solo file, ma vi sono alcune eccezioni. Non dovreste trovare difficoltà a capire in quale file è contenuta la routine che vi interessa:

ISTRUZ.BAS	i listati presentati nel capitolo sul set di istruzioni del BASIC
DOSINTS.BAS	le procedure e le funzioni per richiamare direttamente il Dos dal BASIC
BIOS.BAS	le routine di interfacciamento con il BIOS
TASTIERA.BAS	le funzioni a basso livello della tastiera
MOUSE.BAS	libreria per accedere al mouse
DEMODRAG.BAS	programma dimostrativo del drag-and-drop in BASIC

EMS.BAS	le routine per sfruttare la memoria estesa
OTTIMIZ.BAS	le routine presentate nel capitolo sulla ottimizzazione dei programmi BASIC
TIPS.BAS	le routine presentate nel capitolo di trucchi e suggerimenti
COMPILE.BAT	un semplice programma batch per compilare e linkare un sorgente BASIC
COMPILE2.BAT	revisione del precedente
COMPILE3.BAT	programma per compilare e linkare un programma multi-modulo
FASTCOMP.BAT	esempio di programma per compilare usando un disco virtuale
MYBC.BAS	programma di interfaccia tra l'ambiente di sviluppo e il compilatore
MYBC2.BAS	come il precedente, seconda versione
MYLINK.BAS	programma di interfaccia tra l'ambiente di sviluppo e il linker
BUGS.BAS i listati	che accompagnano la discussione sui bug del BASIC
RICORSIO.BAS	i listati presentati nel capitolo sulla ricorsione
DERIVATE.BAS	il programma per il calcolo delle derivate simboliche
EVALRPN.BAS	un semplice calcolatore in Reverse Polish Notation
EVAL.BAS	il valutatore di espressioni
EVAL.BI	il file di include necessario per richiamare le routine in EVAL.BAS
DEMOEVAL.BAS	un dimostrativo del valutatore di espressioni
SOLVEEQ.BAS	un risolutore di equazioni (richiede EVAL.BAS)
REGINE.BAS	il programma delle otto regine (prima versione)
REGINE2.BAS	il programma per risolvere il problema delle regine e delle amazzoni
REGINE3.EXE	il programma per risolvere il medesimo problema, ottimizzato e riscritto in Assembly
TOTORID.BAS	programma per la riduzione di sistemi di totocalcio

La maggior parte delle routine funzionano nel medesimo modo sotto QuickBASIC, BASIC PDS e Visual BASIC per Dos, anche se in alcuni casi sarà necessario disattivare l'opzione OPTION EXPLICIT di quest'ultimo. Alcuni programmi non possono essere eseguiti con QuickBASIC perché sfruttano caratteristiche del linguaggio assenti in quella versione; in tal caso i commenti nel sorgente spiegano come adattare il programma anche a questa vecchia versione.

I rimanenti file sono dimostrativi di alcuni *tools* per programmatori distribuiti dalla ditta SoftWhale:

DSDEMO.COM	dimostrativo del programma DSWAPPER
DSDEMO.DOC	le istruzioni per usare DSWAPPER
DEMOWIND.EXE	dimostrazione delle dialog box in QBKIT
PULLMENU.EXE	un sistema di menù pulldown multilivello con QBKIT
COLORPIK.EXE	una common dialog box per selezionare i colori in un applicativo
VIRTFORM.EXE	input di dati in un formato virtuale più grande dello schermo
VIEWFILE.EXE	una routine per mostrare il contenuto di un file di testo in una finestra
DEMOSORT.EXE	una dimostrazione della velocità delle routine di ordinamento in QBKIT
C.EXE	Change-Directory utility
SYSINFO.EXE	mostra la configurazione hardware/software
SYSINFO.BW	sorgente del programma precedente, scritto con Batch Wizard
FILEPROC.EXE	programma per manipolare file di testo
FILEPROC.BW	sorgente del programma precedente, scritto con Batch Wizard
CONFIGUR.EXE	permette di modificare in modo interattivo CONFIG.SYS e AUTOEXEC.BAT
CONFIGUR.BW	sorgente del programma precedente, scritto con Batch Wizard

Pur essendo dei dimostrativi, alcuni di questi programmi hanno una discreta utilità; ad esempio VIEWFILE è un velocissimo *browser* per file ASCII, C è un validissimo sostituto del comando CD e compete con il più famoso Norton CD.

FILEPROC può essere usato per eliminare i carriage return in un file di testo in modo da renderlo facilmente editabile con un word processor, e CONFIGUR permette di facilitare le modifiche ai file di configurazione, senza dover editare direttamente CONFIG.SYS e AUTOEXEC.BAT (ottimo per i clienti/utenti alle prime armi); potete usare distribuire questi file senza alcuna restrizione, ma per modificarli avete ovviamente bisogno di BATCH WIZARD, il compilatore per file batch prodotto e distribuito da SoftWhale. Notate la dimensione ridotta dei file eseguibili creati con QBKIT.

PER I PIU' CURIOSI...

Al termine della lettura di questo libro sarete probabilmente d'accordo con me che per scrivere programmi potenti e versatili non è sufficiente conoscere il proprio linguaggio, perché in molti casi la risposta ai problemi si trova nei meandri del Dos e del file system, del BIOS, dell'hardware, ecc.

Molte delle informazioni usate nei programmi che vi ho mostrato sono derivate da testi in lingua inglese, in molti casi quasi introvabili sul mercato italiano.

Se avete un modem e potete collegarvi ad una BBS dovreste eseguire il download di uno dei più famosi prodotti di pubblico dominio; si tratta della *Interrupt List* curata da Ralph Brown con la collaborazione di centinaia di programmatori in tutto il mondo. Questo lungo file di testo indica tutte le possibili funzioni degli interrupt dei sistemi 80x86, compreso le funzione non documentate ufficialmente. C'è davvero tanto da imparare, anche se gli argomenti sono trattati in modo molto schematico.

Per stare al passo con tutte le novità del mondo del BASIC e della programmazione in genere non c'è niente di meglio che una rivista del settore. In Italia l'unica rivista che si rivolge esplicitamente ai programmatori è *Computer Programming*: ogni numero contiene articoli di carattere generale (algoritmi, recensioni di prodotti, ecc.) ed alcune rubriche sui principali linguaggi, tra cui il BASIC e il Visual BASIC per Windows, curate dal sottoscritto. Per ulteriori informazioni potete rivolgervi a:

Infomedia, via Valdera P. 116, 56038 PONSACCO, tel. 0587-735164

Se invece cercate soluzioni pronte, librerie di funzioni per BASIC sotto Dos, DLL o custom control per Visual BASIC per Windows, add-ons, utility o

qualsiasi altro software per abbattere i tempi di sviluppo e migliorare le prestazioni dei vostri programmi e avete bisogno di qualche indicazione o consiglio potete rivolgervi a *Software Design Srl* (tel 080-5045490), in grado di procurarvi qualsiasi prodotto USA in poco tempo.

Francesco Balena

I SEGRETI DI MICROSOFT BASIC

PRINCIPIANTI
INTERMEDI
ESPERTI

GUIDA
REFERENCE
AVANZATO
APPLICATIVO

L'idea alla base di questo libro è di svelare i segreti del BASIC e di approfondire alcuni aspetti trascurati dai manuali del linguaggio e da altri testi sull'argomento. Parte del materiale raccolto è apparso sulla rivista Computer Programming sotto forma di articoli, ma la maggior parte delle informazioni contenute in quest'opera sono inedite. Il libro si rivolge sia a utenti di primo livello che a programmatori esperti, riportando informazioni che consentiranno di risparmiare ore di duro lavoro e aiutando il lettore a creare programmi efficienti e a prova d'errore. Le routine e le tecniche descritte sono applicabili a Microsoft QuickBASIC 4.x, Microsoft BASIC PDS 7.x e Microsoft Visual Basic per DOS.

Tra gli argomenti trattati, meritano speciale menzione:

- Come richiamare direttamente le funzioni del DOS e del BIOS
- I segreti per ottimizzare il codice prodotto dal compilatore
- Come sfruttare il mouse e la memoria espansa EMS
- Numerose procedure già pronte per creare array di bit, ottenere informazioni sul sistema, modificare data e attributi di un file, leggere e scrivere da disco un array in un'unica operazione, e altro ancora
- Tutti gli accorgimenti per ottenere il massimo dal compilatore e dal linker, con la descrizione di un sistema di MAKE integrato all'ambiente di sviluppo per ricompilare solo i moduli che sono stati modificati dall'ultima compilazione
- Un completo Parser e valutatore di espressioni, per risolvere equazioni di qualsiasi grado e tipo
- La descrizione di alcune tecniche avanzate di programmazione, come la ricorsione e il backtraking; sono forniti come esempi un programma che risolve il problema delle regine e una procedura per la riduzione di sistemi per totocalcio
- L'elenco di tutti i bug dell'interprete e del compilatore!

Il dischetto che correda l'opera contiene il sorgente di tutte le routine e di tutti i programmi presentati nel testo, verificati e pronti all'uso.

SOFT WHALE

Francesco Balena è laureato in Scienze dell'Informazione e titolare della SoftWhale, software house specializzata nella produzione di librerie e strumenti per programmatori.

ISBN 88-256-0508-0

Cod.1337 L.49.000